



Uttar Pradesh Rajarshi Tandon  
Open University

# MCS - 113

## Master of Computer Science

### Theory of Computation

## Theory of Computation

<b>Block - 1</b>	<b>Introduction to Finite Automata</b>	<b>3</b>
Unit - 1	Alphabet , Strings and Languages	6
Unit - 2	Finite Automata	26
Unit - 3	Introduction to Machines	48
<b>Block - 2</b>	<b>Regular Expressions and Languages</b>	<b>67</b>
Unit - 4	Regular Expressions	70
Unit - 5	Properties of Regular Language	89
<b>Block - 3</b>	<b>Context Free Grammar</b>	<b>97</b>
Unit - 6	Context Free Grammar	100
Unit - 7	Normal Forms	110
Unit - 8	Context Free Languages ( CFL)	123
<b>Block - 4</b>	<b>Pushdown Automata and Turing Machine</b>	<b>133</b>
Unit – 9	Push Down Automata	136
Unit – 10	Turing Machine	149
Unit – 11	Undecidability	175







Uttar Pradesh Rajarshi Tandon  
Open University

# MCS - 113

## Master of Computer Science

### Theory of Computation

# Block 1

## Introduction to Finite Automata

Unit - 1	
Alphabet, Strings and Languages	6
Unit - 2	
Finite Automata	26
Unit - 3	
Introduction to Machines	48

---

## Course Design Committee

---

<b>Prof. Ashutosh Gupta</b> Director (In-charge) School of Computer and Information Sciences, UPRTOU Prayagraj	<b>Chairman</b>
<b>Prof. R. S. Yadav</b> Department of Computer Science and Engineering MNNIT Prayagraj	<b>Member</b>
<b>Dr. Marisha</b> Assistant Professor (Computer Science), School of Sciences, UPRTOU Prayagraj	<b>Member</b>
<b>Mr. Manoj Kumar Balwant</b> Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	<b>Member</b>

---

## Course Preparation Committee

---

<b>Dr. Ravi Shankar Shukla</b> Associate Professor Department of CSE, Invertis University Bareilly-243006, Uttar Pradesh	<b>Author</b>
<b>Prof. Abhay Saxena</b> Professor and Head, Department of Computer Science Dev Sanskriti Vishwavidyalya, Hardwar, Uttrakhand	<b>Editor</b>
<b>Prof. Ashutosh Gupta</b> Director (In-charge) School of Computer and information, Sciences, UPRTOU Prayagraj	
<b>Mr. Manoj Kumar Balwant</b> Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	<b>Course Coordinator</b>

---

© UPRTOU , Prayagraj - 2023

© MCS - 113 Theory of Computation

ISBN :

---

All Rights are reserved. No Part of this work may reproduced in any form, by mimeograph or any other means, without permission in writing from the Uttar Pradesh Rajarshi Tandon Open University.

Printed and Published by Vinay Kumar Registrar, Uttar Pradesh rajarshi Tandon Open University, Prayagraj - 2023

**Printed By. – M/s K.C.Printing & Allied Works, Panchwati, Mathura -281003.**

---

## Block Introduction

---

At the time of transition, the automata can either move to the next state or stay in the same state. Finite automata have two states, Accept state or Reject state. When the input string is processed successfully, and the automata reached its final state, then it will accept.

Finite automata can be represented by input tape and finite control. Input tape is a linear tape having some number of cells. Each input symbol is placed in each cell. The finite control decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right, and at a time only one input symbol is read.

So we will begin the first unit on Alphabet, Strings and Languages. In this unit firstly we discussed about Set, Relations, Alphabet, Strings, Languages, Finite Representation of Languages, and Chomsky Hierarchy.

Second unit begins with finite automata. In this unit you will know all about basic functionality of finite state system, Basic Definitions of Non-Deterministic finite automata (NFA), Deterministic finite automata (DFA). In this unit you will also learn about the designing of DFA and NFA machines. We will also describe the Equivalence of DFA and NFA. You will learn about Finite automata with epsilon transitions and Removal of epsilon transitions.

In the third unit, we will provide another important topic i.e. Introduction to Machines. In the unit we will describe about the Concept of basic Machine, Properties and limitations of FSM (finite state machine), Moore and mealy Machines. In the Moore and mealy machines, you will learn about the designing of these machines and the conversion from Moore to mealy machine and mealy to Moore machine. We will also describe about the Equivalence of Moore and Mealy machines. The last topic of this unit will be the Minimization of DFA.

As you study the material, you will find that figures, tables are properly used and these will help to understand the concept. There are many sections in the units to easily understand the topic. Every unit has summary and review questions in the end of the unit which will help you to review yourself.

In your study, you will find that every unit has different equal length and your study time will vary for each unit. You will enjoy studying the material and once again wish you all the best for your success.

---

# UNIT-I Alphabet, Strings and Languages

---

## Structure

1.0 Introduction

1.1 Set

1.2 Relations

1.3 Alphabet

1.4 Strings

1.5 Languages

1.6 Finite Representation of Languages

1.6.1 Regular Expressions

1.6.2 Language Represented by a Regular Expression

1.6.3 Regular Languages

1.7 Chomsky Hierarchy

1.8 Summary

1.9 Review Questions

## 1.0 Introduction

This is the first unit of this block. This unit is divided into many sections. Section 1.1 you will learn about Set. Section 1.2 explain Relations. Section 1.3, 1.4 and 1.5 describe Alphabet, Strings and languages respectively. You will also learn about Finite Representation of languages in the section 1.6. This section has some sub sections like Regular Expressions, Language represented by a regular expression and regular languages. In the section 1.7 you will know about Chomsky Hierarchy. Last two section describes summary and Review questions.

### Objectives

After studying this unit, you should be able to:

- Define basics of Finite automata.
- Learn Alphabet, Strings and Languages.
- Describe finite representation of languages and Chomsky Hierarchy

## 1.1 Set

A set is a collection of elements. To indicate that  $x$  is an element of the set  $S$ , we write  $x \in S$ . The statement that  $x$  is not in  $S$  is written as  $x \notin S$ . A set is specified by enclosing some description of its elements in curly braces; for example, the set of all natural numbers  $0, 1, 2, \dots$  denoted by

$$N = \{0, 1, 2, 3, \dots\}$$

We use ellipses (i.e....) When the meaning is clear thus  $J_n = \{1, 2, 3, \dots, n\}$  represents the set of all from 1 to  $n$ .

When the need arises, we use more explicit notation, in which we write

$$S = \{i | i \geq 0, i \text{ is even}\}$$

For the last example. We read this as “ $S$  is the set of all  $i$ , such that  $i$  is greater than zero, and  $i$  is even.”

Considering a “universal set”  $U$ , the complement  $S'$  of  $S$  is defined as

$$S' = \{x | x \in U \wedge x \notin S\}$$

The set with no elements, called the empty set is denoted by  $\emptyset$ . It is obvious that

$$S \cup \emptyset = S - \emptyset = S$$

$$S \cap \emptyset = \emptyset$$

$$\emptyset' = U$$

$$S' = S$$

A set  $S_1$  is said to be a subset of  $S$  if every element of  $S_1$  is also an element of  $S$ . We write this as

$$S_1 \subseteq S$$

If  $S_1 \subseteq S$ , but  $S$  contains an element not in  $S_1$ , we say that  $S_1$  is a proper subset of  $S$ ; we write this as

$$S_1 \subset S$$

The following identities are known as the de Morgan's laws,

$$1. \quad \overline{S_1 \cup S_2} = \overline{S_1} \cap \overline{S_2},$$

$$2. \quad \overline{S_1 \cap S_2} = \overline{S_1} \cup \overline{S_2},$$

$$1. \quad \overline{S_1 \cup S_2} = \overline{S_1} \cap \overline{S_2}$$

$$x \in \overline{S_1 \cup S_2}$$

$$\Leftrightarrow x \in U \text{ and } x \notin S_1 \cup S_2$$

$$\Leftrightarrow x \in U \text{ and } \neg(x \in S_1 \text{ or } x \in S_2) \quad (\text{def. Union})$$

$$\Leftrightarrow x \in U \text{ and } (\neg(x \in S_1) \text{ and } \neg(x \in S_2)) \quad (\text{negation of disjunction})$$

$$\Leftrightarrow x \in U \text{ and } (x \notin S_1 \text{ and } x \notin S_2)$$

$$\Leftrightarrow (x \in U \text{ and } x \notin S_1) \text{ and } (x \in U \text{ and } x \notin S_2)$$

$$\Leftrightarrow (x \in \overline{S_1} \text{ and } x \in \overline{S_2}) \text{ (def.complement)}$$

$$\Leftrightarrow x \in \overline{S_1} \cap \overline{S_2} \text{ (def. Intersection)}$$

If  $S_1$  and  $S_2$  have no common element, that is,

$$S_1 \cap S_2 = \emptyset,$$

Then the sets are said to be disjoint.

A set is said to be finite if it contains a finite number of elements; otherwise it is infinite. The size of a finite set is the number of elements in it; this is denoted by  $|S|$  (is or called  $\#S$ ). the power set of  $S$  and is A set may have many subsets. The set of all subsets of a set denoted by  $2^S$  or  $P(S)$ . Observe that  $2^S$  is a set of sets.

### Example

If  $S$  is the set  $\{1,2,3\}$ , then its power set is

$$2^S = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$

Here  $|S| = 3$  and  $|2^S| = 8$ . This is an instance of a general result, if  $S$  is finite, then

$$|2^S| = 2^{|S|}$$

**Proof:** (By induction on the number of elements in  $S$ ).

**Basis:**  $|S| = 1 \Rightarrow 2^S = \{\emptyset, S\} \Rightarrow |2^S| = 2^1 = 2$

**Induction Hypothesis:** Assume the property holds for all sets  $S$  with  $k$  elements.

**Induction Step:** Show that the property holds for (all sets with)  $k + 1$  elements. Denote

$$\begin{aligned} S_{k+1} &= \{y_1, y_2, \dots, y_{k+1}\} \\ &= S_k \cup \{y_{k+1}\} \end{aligned}$$

Where  $S_k = \{y_1, y_2, y_3, \dots, y_k\}$

$$\begin{aligned} 2^{S_{k+1}} &= 2^{S_k \cup \{y_{k+1}\}} \\ &= \{y_1, y_{k+1}\} \cup \{y_2, y_{k+1}\} \cup \dots \cup \{y_k, y_{k+1}\} \cup \\ &\quad \cup_{x, y \in S_k} \{x, y, y_{k+1}\} \cup \dots \\ &\quad \cup S_{k+1} \end{aligned}$$

$2^{S_k}$  has  $2^k$  elements by the induction hypothesis.

The number of sets in  $2^{S_{k+1}}$  which contain  $y_{k+1}$  is also  $2^k$ .

Consequently  $|2^{S_{k+1}}| = 2 * 2^k = 2^{k+1}$ .

A set, which has as its elements ordered sequences of elements from other sets, is called the Cartesian product of the other sets. For the Cartesian product of two sets, which itself is a set of ordered pairs, we write

$$S = S_1 \times S_2 = \{(x,y) \mid x \in S_1, y \in S_2\}$$

**Example:**

Let  $S_1 = \{1,2\}$  and  $S_2 = \{1,2,3\}$ . Then

$$S_1 \times S_2 = \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3)\}$$

Note that the order in which the elements of a pair are written matters; the pair (3,2) is not in  $S_1 \times S_2$ .

**Example:**

If A is the set of throws of a coin, i.e.,  $A = \{\text{head}, \text{tail}\}$ , then

$$A \times A = \{(\text{head}, \text{head}), (\text{head}, \text{tail}), (\text{tail}, \text{head}), (\text{tail}, \text{tail})\}$$

the set of all possible throws of two coins.

The notation is extended in an obvious fashion to the Cartesian product of more than two sets; generally

$$S_1 \times S_2 \times \cdots \times S_n = \{(x_1, x_2, \dots, x_n) \mid x_i \in S_i\}$$

## 1.2 Relations

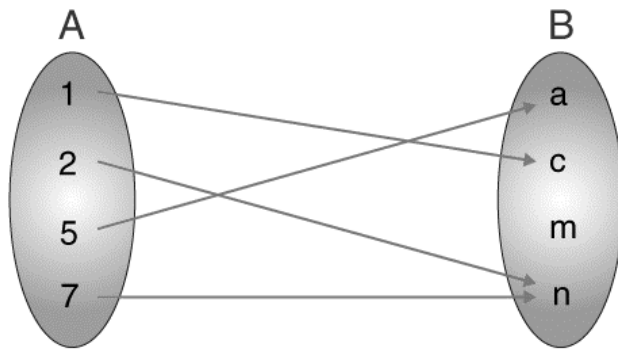
A relation in mathematics defines the relationship between two different sets of information. If two sets are considered, the relation between them will be establish, if there is a connection between the elements of two or more non-empty sets.

In the morning assembly at schools, students are supposed to stand in a queue in ascending order of the heights of all the students. This defines an ordered relation between the students and their heights.

Therefore, we can say,

‘A set of ordered pairs is defined as a relation.’





**Figure 1.1: Relation**

This mapping depicts a relation from set A into set B. A relation from A to B is a subset of  $A \times B$ . The ordered pairs are  $(1, c)$ ,  $(2, n)$ ,  $(5, a)$ ,  $(7, n)$ . For defining a relation, we use the notation where,

Set  $\{1, 2, 5, 7\}$  represents the domain.

Set  $\{a, c, n\}$  represents the range.

Sets and relation are interconnected with each other. The relation defines the relation between two given sets.

If there are two sets available, then to check if there is any connection between the two sets, we use relations.

For example, an empty relation denotes none of the elements in the two sets is same.

In Mathematics, the relation is the relationship between two or more set of values.

Suppose, x and y are two sets of ordered pairs. And set x has relation with set y, then the values of set x are called domain whereas the values of set y are called range.

### **Example:**

For ordered pairs=  $\{(1,2), (-3,4), (5,6), (-7,8), (9,2)\}$

The domain is =  $\{-7, -3, 1, 5, 9\}$

And range is =  $\{2, 4, 6, 8\}$

## Types of Relations

There are 8 main types of relations which include:

- Empty Relation
- Universal Relation
- Identity Relation
- Inverse Relation
- Reflexive Relation
- Symmetric Relation
- Transitive Relation
- Equivalence Relation

### Empty Relation

An empty relation (or void relation) is one in which there is no relation between any elements of a set. For example, if set  $A = \{1, 2, 3\}$  then, one of the void relations can be  $R = \{x, y\}$  where,  $|x - y| = 8$ . For empty relation,

$$R = \varnothing \subset A \times A$$

### Universal Relation

A universal (or full relation) is a type of relation in which every element of a set is related to each other. Consider set  $A = \{a, b, c\}$ . Now one of the universal relations will be  $R = \{x, y\}$  where,  $|x - y| \geq 0$ . For universal relation,

$$R = A \times A$$

### Identity Relation

In an identity relation, every element of a set is relating to itself only. For example, in a set  $A = \{a, b, c\}$ , the identity relation will be  $I = \{a, a\}, \{b, b\}, \{c, c\}$ . For identity relation,

$$I = \{(a, a), a \in A\}$$

### Inverse Relation

Inverse relation is seen when a set has elements which are inverse pairs of another set. For example, if set  $A = \{(a, b), (c, d)\}$ , then inverse relation will be  $R^{-1} = \{(b, a), (d, c)\}$ . So, for an inverse relation,

$$R^{-1} = \{(b, a): (a, b) \in R\}$$

### Reflexive Relation

In a reflexive relation, every element maps to itself. For example, consider a set  $A = \{1, 2\}$ . Now an example of reflexive relation will be  $R = \{(1, 1), (2, 2), (1, 2), (2, 1)\}$ . The reflexive relation is given by-

$$(a, a) \in R$$

### Symmetric Relation

In a symmetric relation, if  $a=b$  is true then  $b=a$  is also true. In other words, a relation  $R$  is symmetric only if  $(b, a) \in R$  is true when  $(a,b) \in R$ . An example of symmetric relation will be  $R = \{(1, 2), (2, 1)\}$  for a set  $A = \{1, 2\}$ . So, for a symmetric relation,

$$aRb \Rightarrow bRa, \forall a, b \in A$$

### Transitive Relation

For transitive relation, if  $(x, y) \in R$ ,  $(y, z) \in R$ , then  $(x, z) \in R$ . For a transitive relation,

$$aRb \text{ and } bRc \Rightarrow aRc \forall a, b, c \in A$$

### Equivalence Relation

If a relation is reflexive, symmetric and transitive at the same time it is known as an equivalence relation.

## 1.3 Alphabet

An alphabet, in the context of formal language theory, is a finite non-empty set. Typically, it is denoted  $\Sigma$  or  $V$  (where  $V$  stands for vocabulary). Examples range from the binary alphabet  $\{0,1\}$  to the keywords for a particular programming language.

#### Examples:

$\Sigma = \{0, 1\}$  is an alphabet of binary digits  
 $\Sigma = \{A, B, C, \dots, Z\}$  is an alphabet.

The components of an alphabet are stated as the letters (or symbols) of the alphabet. With the help of an alphabet we may acquire strings (or words) which are sequences of finite letters over  $\Sigma$ . The empty string, denoted  $\lambda$  or  $\epsilon$ , is also deliberated as a string having no letters.

The set of all words over  $\Sigma$  (including the empty word) is symbolized by  $\Sigma^*$  and is denoted as the Kleene star (or closure) of  $\Sigma$  (or monoid closure) after the American Mathematician S. C. Kleene. The set  $\Sigma^* \setminus \{\lambda\}$  is denoted  $\Sigma^+$  and is referred to as the Kleene plus (or semigroup closure) of  $\Sigma$ . The names

monoid and semigroup closure being justified by  $\Sigma^*$  and  $\Sigma^+$  forming a monoid and semi-group under concatenation respectively.

## 1.4 Strings

A string is a finite sequence of symbols selected from some alphabet. It is generally denoted as  $w$ . For example, for alphabet  $\Sigma = \{0, 1\}$   $w = 010101$  is a string.

Length of a string is denoted as  $|w|$  and is defined as the number of positions for the symbol in the string. For the above example length is 6.

The empty string is the string with zero occurrence of symbols. This string is represented as  $\epsilon$  or  $\lambda$ .

The set of strings, including the empty string, over an alphabet  $\Sigma$  is denoted by  $\Sigma^*$ .

For  $\Sigma = \{0, 1\}$  we have set of strings as  $\Sigma^* = \{\epsilon, 0, 1, 01, 10, 00, 11, 10101\dots\}$ . And  $\Sigma^1 = \{0, 1\}$ ,  $\Sigma^2 = \{00, 01, 10, 11\}$  and so on.

$\Sigma^*$  contains an empty string  $\epsilon$ . The set of non- empty string is denoted by  $\Sigma^+$ . From this we get:

$$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$$

### Operations on Strings

#### 8) Length of a String:

- Definition – It is the number of symbols present in a string. (Denoted by  $|S|$ ).
- Examples –
  - If  $S = \text{'cabcad'}$ ,  $|S| = 6$
  - If  $|S| = 0$ , it is called an empty string (Denoted by  $\lambda$  or  $\epsilon$ )

#### 2) Substring:

- Definition – A sequence of symbols from any part of the given string over an alphabet is called a “substring of a string”.
- Examples –

For string  $abb$  over  $\Sigma = \{a,b\}$ . The passive substrings are:

- Zero length substring:  $\epsilon$
- One length substring:  $a, b$
- Two length substring:  $ab, aa$

- Three length substring: aab, baa

### 3) Concatenation:

- Definition – combines two strings by putting them one after the other.
- Example –  $x = abc$ ,  $y = mnop$ , then  $x \circ y = abcmnop$ , or simply  $xy = abcmnop$

The concatenation of the empty string with any other string gives the string itself:  $x e = ex = x$

### 4) Reversal:

- Definition – Reversal of a string  $w$  denoted  $w^R$  is the string spelled backwards
- Formal definition:
  - If  $w$  is a string of length 0, then  $w^R = w = e$
  - If  $w$  is a string of length  $n+1 > 0$ , then  $w = ua$  for some  $a \in \Sigma$ , and  $w^R = a u^R$ .

### 5) Prefix of a string:

- Definition – A substring with the sequence of beginning symbols of a given string is called a “prefix”.
- Example –

For a string  $abb$ , the possible prefixes of  $abb$  are:

- $\epsilon$  (zero length prefix)
- $a$  (one length prefix)
- $ab$  (two length prefix)
- $abb$  (three length prefix)

### 6) Suffix of a string:

- Definition – A substring with the sequence of ending symbols of a given string is called a “suffix”.
- Example –

For a string  $abb$ , the possible suffixes of  $abb$  are:

- $\epsilon$  (zero length suffix)
- $b$  (one length suffix)
- $ba$  (two length suffix)
- $bba$  (three length suffix)

### 7) Proper prefix of a string:

- Definition – Proper prefix is a prefixes except the given string.
- Example –

For a string abb, the possible proper prefixes are:  $\epsilon$ , a, ab.

### 8) Proper suffix of a string:

- Definition – Proper suffix is a suffix except the given string.
- Example –

For a string abb, the possible proper suffix is:  $\epsilon$ , b, ba.

---

## Check your progress

---

Q1. What is set in theory of computation? Explain with example.

Q2. Define the term relation in theory of computation?

Q3. How many strings of length 2 and starting with “a” are possible over alphabet  $\{a,b\}$ ?

## 1.5 Language

The language, that is used with a regular expression or deterministic finite automata or non-deterministic finite automata or a state machine is known as a regular language.

In other words, we can say that a language is a set of strings. These strings are created with characters from specified alphabet, or set of symbols. From these strings we can get Regular language. In this case we can say that the subset of the set of all string is known as regular language. This regular language can be used in parsing and designing programming languages. These languages are beneficial for facilitating computer scientists to identify patterns in data and group certain computational problems together — once they do that, they can take similar approaches to solve the problems grouped together. Regular languages are very important and useful topic in computability theory.

We start with a finite, nonempty set  $\Sigma$  of symbols, called the alphabet. From the individual symbols we construct strings (over  $\Sigma$  or on  $\Sigma$ ), which are finite sequences of symbols from the alphabet. The empty string  $\varepsilon$  is a string with no symbols at all. Any set of strings over/on  $\Sigma$  is a language over/on  $\Sigma$ .

**Example:**

$$\begin{aligned}\Sigma &= \{c\} \\ L1 &= \{cc\} \\ L2 &= \{c, cc, ccc\} \\ L3 &= \{w \mid w = ck, k = 0, 1, 2, \dots\} \\ &= \{\varepsilon, c, cc, ccc, \dots\}\end{aligned}$$

**Example:**

$$\begin{aligned}\Sigma &= \{a, b\} \\ L1 &= \{ab, ba, aa, bb, \varepsilon\} \\ L2 &= \{w \mid w = (ab)k, k = 0, 1, 2, 3, \dots\} \\ &= \{\varepsilon, ab, abab, ababab, \dots\}\end{aligned}$$

The concatenation of two strings  $w$  and  $v$  is the string obtained by appending the symbols of  $v$  to the right end of  $w$ , that is, if

$$w = a_1a_2\dots a_n$$

and

$$v = b_1b_2\dots b_m,$$

then the concatenation of  $w$  and  $v$ , denoted by  $wv$ , is

$$wv = a_1a_2\dots a_nb_1b_2\dots b_m$$

If  $w$  is a string, then  $w^n$  is the string obtained by concatenating  $w$  with itself  $n$  times. As a special case, we define

$$w^0 = \varepsilon,$$

for all  $w$ . Note that  $\varepsilon w = w\varepsilon = w$  for all  $w$ . The reverse of a string is obtained by writing the symbols in reverse order; if  $w$  is a string as shown above, then its reverse  $w^R$  is

$$w^R = a_n\dots a_2a_1$$

If

$$w = uv,$$

then  $u$  is said to be prefix and  $v$  a suffix of  $w$ .

The length of a string  $w$ , denoted by  $|w|$ , is the number of symbols in the string. Note that,

$$|\varepsilon| = 0$$

If  $u$  and  $v$  are strings, then the length of their concatenation is the sum of the individual lengths,

$$|uv| = |u| + |v|$$

Let us show that  $|uv| = |u| + |v|$ . To prove this by induction on the length of strings, let us define the length of a string recursively, by

$$\begin{aligned} |a| &= 1 \\ |wa| &= |w| + 1 \end{aligned}$$

For all  $a \in \Sigma$  and  $w$  any string on  $\Sigma$ . This definition is a formal statement of our intuitive understanding of the length of a string: the length of a single symbol is one, and the length of any string is incremented by one if we add another symbol to it.

**Basis**  $|uv| = |u| + |v|$  holds for all  $u$  of any length and all  $v$  of length 1 (by definition).

**Induction Hypothesis:** we assume that  $|uv| = |u| + |v|$  holds for all  $u$  of any length and all  $v$  of length  $1, 2, \dots, n$ .

**Induction Step:** Take any  $v$  of length  $n + 1$  and write it as  $v = wa$ . Then,

$$\begin{aligned} |v| &= |w| + 1, \\ |uv| &= |uwa| = |uw| + 1. \end{aligned}$$

By the induction hypothesis (which is applicable since  $w$  is of length  $n$ ).

$$|uw| = |u| + |w|.$$

So that

$$|uv| = |u| + |w| + 1 = |u| + |v|$$

Which completes the induction step?

If  $\Sigma$  is an alphabet, then we use  $\Sigma^*$  to denote the set of strings obtained by concatenating zero or more symbols from  $\Sigma$ . We denote  $\Sigma^+ = \Sigma^* - \{\epsilon\}$ .

The sets  $\Sigma^*$  and  $\Sigma^+$  are always infinite.

A language can thus be defined as a subset of  $\Sigma^*$ . A string  $w$  in a language  $L$  is also called a word or a sentence of  $L$ .



**Example:**

$\Sigma = \{a,b\}$ . Then

$$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}.$$

The set

$$\{a, aa, aab\}.$$

Is a language on  $\Sigma$ . Because it has a finite number of words, we call it a finite language. The set

$$L = \{a^n b^n \mid n \geq 0\}$$

is also a language on  $\Sigma$ . The strings  $aabb$  and  $aaaabbbb$  are words in the language  $L$ , but the string  $abb$  is not in  $L$ . This language is infinite.

Since languages are sets, the union, intersection, and difference of two languages are immediately defined. The complement of a language is defined with respect to  $\Sigma^*$ ; that is, the complement of  $L$  is

$$L' = \Sigma^* - L$$

The concatenation of two languages  $L_1$  and  $L_2$  is the set of all strings obtained by concatenating any element of  $L_1$  with any element of  $L_2$ ; specifically,

$$L_1 L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$$

We define  $L^n$  as  $L$  concatenated with itself  $n$  times, with the special case

$$L^0 = \{\epsilon\}$$

for every language  $L$ .

## 1.6 Finite Representation of Language

Languages may be infinite sets of strings. We need a finite notation for them.

There are at least four ways to do this:

1. Language generators: The language can be represented as a mathematical sequence  $w_1, w_2, w_3, \dots$  such that the language is equal to the set  $\{w_1, w_2, w_3, \dots\}$ . Given an integer  $i$ , the generator will produce the string  $w_i$ .
2. Language acceptors: The language can be represented as a mathematical predicate, a membership tester. Given a string, this will tell if the string is in the language.
3. Mathematical descriptions, like  $\{a^n b^n : n \geq 0\}$ .
4. Explicit listings, like  $\{0, 1, 00, 01\}$ .

Some other ways are also noticed.

- Explicit listings work only for finite languages.
- Math descriptions are very general, but it may be hard to know if a string is in the language.
- Language acceptors have a hard time answering some questions, such as, whether the language is empty.
- Language generators have a hard time testing if a string is in the language.

There are uncountable many languages over a nonempty set  $\Sigma$  but only countable many representations in a finite set of symbols. Therefore, most languages will never have a finite representation.

### 1.6.1 Regular Expressions

Regular expressions are one way to represent languages. They are analogous to arithmetic expressions for representing quantities. This notation will turn out to be useful for describing programming languages and also for text searching applications.

There are rules of inference for constructing regular expressions over an alphabet  $\Sigma$ .

- If  $a \in \Sigma$  then  $a$  itself is a regular expression over  $\Sigma$ .
- $\emptyset$  is a regular expression over  $\Sigma$ .
- If  $E$  and  $F$  are regular expressions over  $\Sigma$  then so is  $(EF)$ .
- If  $E$  and  $F$  are regular expressions over  $\Sigma$  then so is  $(E \cup F)$ .
- If  $E$  is a regular expression over  $\Sigma$  then so is  $(E^*)$ .
- Parentheses can often be omitted.

**Example:** Suppose  $\Sigma = \{0,1\}$ .

Then  $0$  is a regular expression over  $\{0, 1\}$  by 1.

So  $(0^*)$  is a regular expression over  $\{0, 1\}$  by 5.

Also,  $1$  is a regular expression over  $\{0, 1\}$  by 1.

So  $1(0^*)$  is a regular expression over  $\{0, 1\}$  by 3.

Also  $(1^*)$  is a regular expression over  $\{0, 1\}$  by 5.

So  $0(1^*)$  is a regular expression over  $\{0, 1\}$  by 3.

Thus  $1(0^*) \cup 0(1^*)$  is a regular expression over  $\{0, 1\}$  by 4

This regular expression represents the language  $(\{1\}\{0\}^*) \cup (\{0\}\{1\}^*)$ . This language contains strings like  $\{1, 10, 100, 1000, \dots, 0, 01, 011, 0111, \dots\}$ . Note that  $\{0, 1\}^*$  is not a regular expression over the alphabet  $\{0, 1\}$ .

### 1.6.2 Language Represented by a Regular Expression

If  $E$  is a regular expression, then let  $L(E)$  be the language it represents. We have the following rules:

$$\text{If } a \in \Sigma \text{ then } L(a) = \{a\}.$$

$$L(\emptyset) = \emptyset$$

$$L(EF) = L(E) \circ L(F)$$

$$L(E \cup F) = L(E) \cup L(F)$$

$$L(E^*) = L(E)^*$$

Note that  $L(E) \circ L(F)$  is the concatenation of two languages,  $L(E) \cup L(F)$  is the union of two languages, and  $L(E)^*$  is the Kleene star of a language. Thus for example

$$\begin{aligned} L(1(0^*) \cup 0(1^*)) &= \\ L(1(0^*)) \cup L(0(1^*)) &= \\ (L(1) \circ L(0^*)) \cup (L(0) \circ L(1^*)) &= \\ (\{1\} \circ \{0\}^*) \cup (\{0\} \circ \{1\}^*). \end{aligned}$$

### 1.6.3 Regular Languages

A language  $L$  is said to be regular if there is a regular expression  $E$  such that  $L = L(E)$ , that is, if  $L$  can be represented by a regular expression.

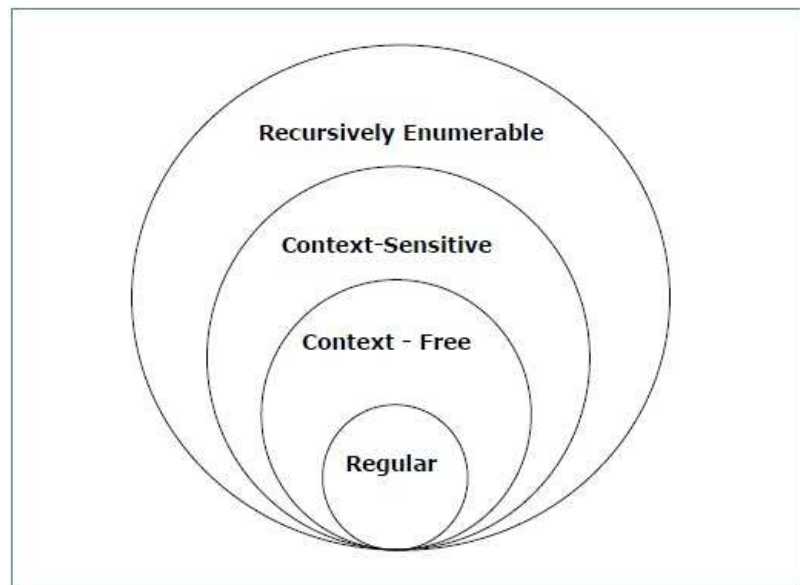
## 1.7Chomsky Hierarchy

According to Noam Chomsky, there are four types of grammars – Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other –

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

**Table 1: Chomsky Hierarchy Table**

Take a look at the following illustration. It shows the scope of each type of grammar –



**Figure 1.2: Chomsky Hierarchy**

### Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form  $X \rightarrow a$  or  $X \rightarrow aY$

Where  $\mathbf{X, Y \in N}$  (Non terminal) and  $\mathbf{a \in T}$  (Terminal)

The rule  $\mathbf{S \rightarrow \epsilon}$  is allowed if  $\mathbf{S}$  does not appear on the right side of any rule.

**Example**

$$X \rightarrow \epsilon$$

$$X \rightarrow a \mid aY$$

$$Y \rightarrow b$$

**Type - 2 Grammar**

Type-2 grammars generate context-free languages. The productions must be in the form  $\mathbf{A \rightarrow \gamma}$  where  $\mathbf{A \in N}$  (Non terminal) and  $\mathbf{\gamma \in (T \cup N)^*}$  (String of terminals and non-terminals).

These languages generated by these grammars are recognized by a non-deterministic pushdown automaton.

**Example:**

$$S \rightarrow Xa$$

$$X \rightarrow a$$

$$X \rightarrow aX$$

$$X \rightarrow abc$$

$$X \rightarrow \epsilon$$

**Type - 1 Grammar**

Type-1 grammars generate context-sensitive languages. The productions must be in the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

Where  $\mathbf{A \in N}$  (Non-terminal) and  $\mathbf{\alpha, \beta, \gamma \in (T \cup N)^*}$  (Strings of terminals and non-terminals)

The strings  $\alpha$  and  $\beta$  may be empty, but  $\gamma$  must be non-empty.

The rule  $\mathbf{S \rightarrow \epsilon}$  is allowed if  $\mathbf{S}$  does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

**Example:**

$$AB \rightarrow AbBc$$

$$A \rightarrow bcA$$

$$B \rightarrow b$$

## Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of  $\alpha \rightarrow \beta$  where  $\alpha$  is a string of terminals and non-terminals with at least one non-terminal and  $\alpha$  cannot be null.  $\beta$  is a string of terminals and non-terminals.

### Example:

$$S \rightarrow ACaB$$

$$Bc \rightarrow acB$$

$$CB \rightarrow DB$$

$$aD \rightarrow Db$$

---

## 1.8 Summary

---

In this unit you have learnt about Set, Relations, Alphabet, Strings, Languages, Finite Representation of Languages, and Chomsky Hierarchy.

- A set is said to be finite if it contains a finite number of elements; otherwise it is infinite.
- The size of a finite set is the number of elements in it; this is denoted by  $|S|$  (or  $\#S$ ). A set may have many subsets.
- The set of all subsets of a set  $S$  is called the power set of  $S$  and is denoted by  $2S$  or  $P(S)$ .
- A relation between two sets is a collection of ordered pairs containing one object from each set. If the object  $x$  is from the first set and the object  $y$  is from the second set, then the objects are said to be related if the ordered pair  $(x,y)$  is in the relation.
- A finite Language can be representative by exhaustive enumeration of all the string in the languages.

---

## 1.9 Review Questions

---

- Q1. Where are sets used in theory of computation? Elaborate your answer.
- Q2. Define the relations in theory of computation with suitable example.
- Q3. How many strings of length less than 4 contains the language?
- Q4. Show that for any language  $L$ ,  $L^* = (L^*)^* = (L^+)^* = (L^*)^+$
- Q5. Describe the language corresponding to following:  $(1+01)^*(0+01)^*$
- Q6. Find a grammar generating  $L = \{a_n b_n c_i\} n \geq 1, i \geq 0\}$
- Q7. Construct a grammar which generates all even integers up to 998.
- Q8. If each production in a grammar  $G$  has some variables on its right hand side, what can you say about  $L(G)$ ?

---

## UNIT-2 Finite Automata

---

### Structure

2.0 Introduction

2.1 Finite State Systems

2.2 Basic Definitions Non-Deterministic finite automata (NFA)

2.3 Deterministic finite automata (DFA)

2.4 Equivalence of DFA and NFA

2.5 Finite automata with epsilon transitions

2.6 Removal of epsilon transitions.

2.7 Summary

2.8 Review Questions



## 2.0 Introduction

Finite Automata is the second unit of this block. In this unit, there are eight sections. First section i.e. section 2.1, describe finite state systems. Section 2.2 explain about basic definition of Non-Deterministic finite automata (NFA). Next section 2.3 is Deterministic finite automata (DFA). In the next section, i.e. section 2.4, you will study about Equivalence of DFA and NFA. Finite automata with epsilon transitions describe in the section 2.5. There is an important section, 2.6 describe Removal of epsilon transitions. Last two sections describe summary and review questions.

### Objectives

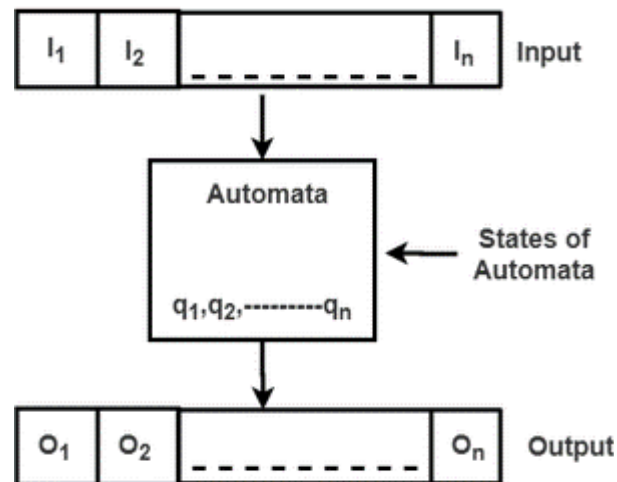
After studying this unit, you should be able to:

- Understand Finite State Systems
- Basics of Non-Deterministic finite automata (NFA), Deterministic finite automata (DFA).
- Equivalence of DFA and NFA.
- Finite automata with epsilon transitions.
- Removal of epsilon transitions.

## 2.1 Finite State Systems

A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. There are two types of state machine, one is deterministic finite-state machines and second is non-deterministic finite-state machines. A deterministic finite-state machine can be built corresponding to any non-deterministic one.

Finite Automata(FA) is the simplest machine to recognize patterns. The finite automata or finite state machine is an abstract machine which have five elements or tuple. It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Basically, it is an abstract model of digital computer. Following figure shows some essential features of a general automation.



**Figure 2.1: Features of Finite Automata**

The above figure shows following features of automata:

- Input
- Output
- States of automata
- State relation
- Output relation

A Finite Automata consists of the following:

*Q: Finite set of states.*

$\Sigma$ : set of Input Symbols.

$q$ : Initial state.

$F$ : set of Final States.

$\delta$ : Transition Function.

Formal specification of machine is

$\{Q, \Sigma, q, F, \delta\}$ .

## 2.2 Non-Deterministic finite automata

NFA is a state machine consisting of states and transitions that can either accept or reject a finite string. And like a DFA, we must use circles to represent states, and directed arrows to represent transitions.

### Formal Definition of an NFA

The formal definition of an NFA consists of a 5-tuple, in which order matters.

Similar to a DFA, the formal definition of NFA is:  $(Q, \Sigma, \delta, q_0, F)$ , where

1.  $Q$  is a finite set of all states
2.  $\Sigma$  is a finite set of all symbols of the alphabet
3.  $\delta: Q \times \Sigma \rightarrow Q$  is the transition function from state to state
4.  $q_0 \in Q$  is the start state, in which the start state must be in the set  $Q$
5.  $F \subseteq Q$  is the set of accept states, in which the accept states must be in the set  $Q$

For our NFA above, the formal definition would be:

- $Q \rightarrow \{s, f\}$
- $\Sigma \rightarrow \{a, b\}$
- Start state  $\rightarrow s$
- $F \rightarrow \{f\}$
- $\delta$  functions:

$$\delta(s, a) = \{f\}$$

$$\delta(s, b) = \{\}$$

$$\delta(s, \epsilon) = \{\}$$

$$\delta(f, a) = \{f\}$$

$$\delta(f, b) = \{f\}$$

$$\delta(f, \epsilon) = \{\}$$

## Graphical Representation of an NFA

An NFA can be represented by digraphs called state Diagram, In which:

- The state is represented by vertices.
- The arc labelled with an input character show the transitions.
- The initial state is marked with an arrow.
- The final state is denoted by the double circle.

### Example:

$$Q = \{q_0, q_1, q_2\}$$

$$\Sigma = \{0, 1\}$$

$$q_0 = \{q_0\}$$

$$F = \{q_2\}$$

### Solution:

Transition diagram:

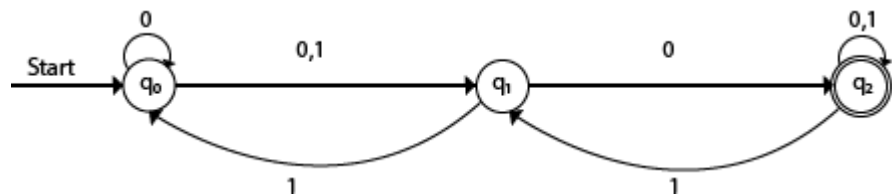


Figure 2.2: NFA

### Transition Table:

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	$q_0, q_1$	$q_1$
$q_1$	$q_2$	$q_0$
$*q_2$	$q_2$	$q_1, q_2$

Table 2: Transition Table

In the above diagram, we can see that when the current state is  $q_0$ , on input 0, the next state will be  $q_0$  or  $q_1$ , and on 1 input the next state will be  $q_1$ . When the current state is  $q_1$ , on input 0 the next state will be  $q_2$  and on 1 input, the next state will be  $q_0$ . When the current state is  $q_2$ , on 0 input the next state is  $q_2$ , and on 1 input the next state will be  $q_1$  or  $q_2$ .

## 2.3 Deterministic finite automata (DFA)

A DFA is a state machine consisting of states and transitions that can either accept or reject a finite string, which consists of a series of symbols, and compare it to a predefined language across a predetermined set of characters. We use circles to represent states, and directed arrows to represent transitions. Every state must have each symbol going outwards from the state, or else it will not be defined as a DFA.

DFAs allow for an easier use of certain projects and applications that switch between states of validity and invalidity. DFAs are useful in the functionality of applications such as:

- Speech recognition and processing
- Pattern matching
- Applications that consist of some sort of on/off functionality
- Compilers for modern programming languages

### Formal definition of a DFA

The formal definition of a DFA consists of a 5-tuple, in which order matters.

The formal definition of DFA is:  $(Q, \Sigma, \delta, q_0, F)$ , where

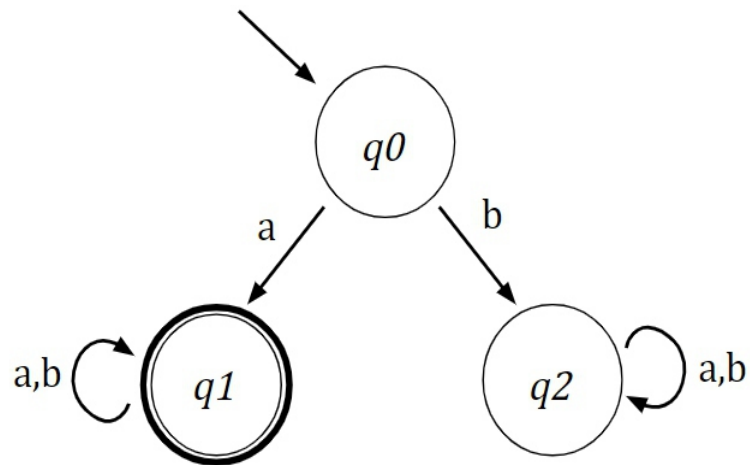
Let's take a look at what a DFA actually looks like. Suppose we wanted to only accept strings that begin with the letter a. For simplicity, our language will only consist of the characters  $\{a, b\}$ . So, what would our language look like?

It would look something like this-

$$\{ax \mid x \in \{a, b\}^*\}$$

This language states that we will only accept strings that begin with a, such that x consists of the language  $\{a, b\}$ . The star (\*), more specifically known as the Kleene star, specifies that x can consist of any order of characters, any number of times, of the alphabet  $\{a, b\}$ . So, if you look back at our language, we are specifically stating, "our string MUST begin with an a, and can be followed by literally anything after that." We don't care what comes after the initial a.

Here is a DFA for this language...

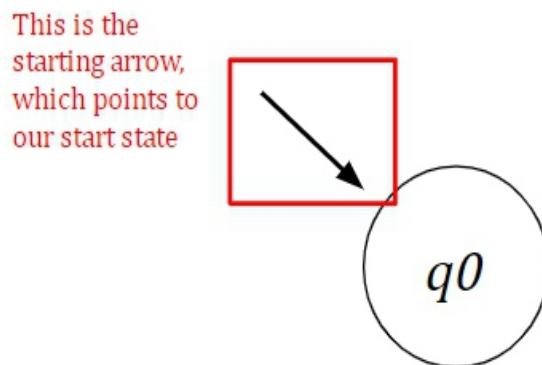


**Figure 2.3: DFA**

### How do we draw a DFA? (Components of a DFA) [1]

Let's take a look at our DFA and look at each component to see what is actually going on.

First we must begin with the initial arrow.

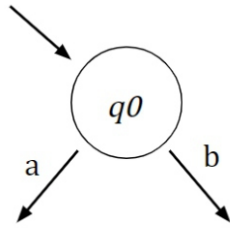


**Figure 2.4: Starting arrow**

The starting arrow is the first thing we must look at in our DFA. This arrow points to our start state. Now that we know our start state, we can begin tracing the occurrences of each symbol.

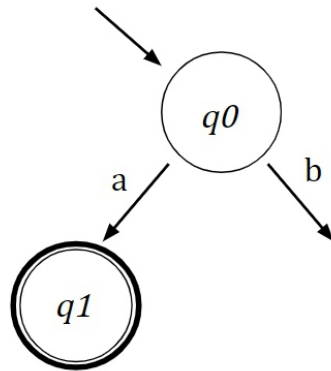
The start state cannot be an accepting state, because that would imply that an empty string ( $\epsilon$ ) would accept the language. However, because we need at least an  $a$  in the string, an empty string cannot be valid.

Our language consists of the symbols  $\{a, b\}$ , our start state can only transition on those two symbols. So, we have...



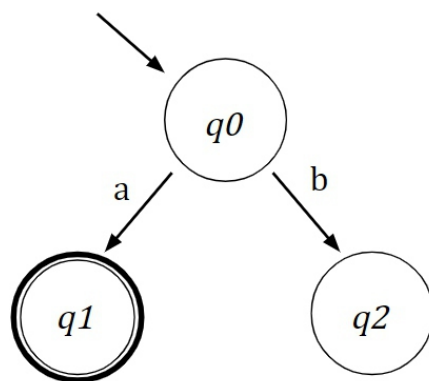
**Figure 2.5: Transition on two symbols**

From this position, if we get an  $a$  in the beginning of the string, that means we can accept the string. An accepting state is depicted by two double circles. See below...



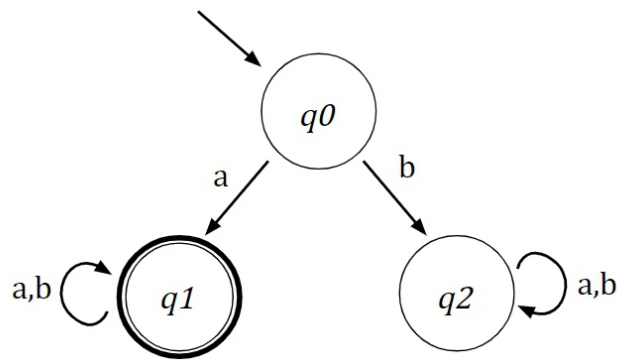
**Figure 2.6: An accepting state is depicted by two double circles**

If we get the letter  $b$  as the first symbol in the string, we must reject the string. See below...



**Figure 2.7: Reject the string in case of  $b$**

We are still not finished yet. We still must state in our DFA that, in our accepting state, anything can come after the initial  $a$  in our string, because we only care that  $a$  was the initial string. Also, let's say we get a  $b$  as our first character, we must also state that any symbol after that must also reject the string, because the string will always start with  $b$ . See below.



**Figure 2.8: The complete DFA**

We have now successfully created a DFA that only accepts strings that begin with the symbol  $a$ .

---

## Check your progress

---

- Q1. What are the tuples of NFA? Define with example.
- Q2. How do you create a DFA in automata? Elaborate your answer.
- Q3. Compare and contrast NFA and DFA with a suitable example.

## 2.4 Equivalence of DFA and NFA [2]

The term Finite state automata (FSA) is also known as finite state machines (FSM). The finite state machines are generally categorized as being deterministic (DFA) or non-deterministic (NFA). A deterministic finite state automaton has exactly one transition from every state for each possible input. In other words, whatever state the FSA is in, if it encounters a symbol for which a transition exists, there will be just one transition and obviously as a result, one follows up state. For a given string, the path through a DFA is deterministic since there is no place along the way where the machine would have to choose between more than one transitions.

Given this definition, it isn't too hard to figure out what an NFA is. Unlike in DFA, it is possible for states in an NFA to have more than one transition per input symbol. Additionally, states in an NFA may have states that don't require an input symbol at all, transitioning on the empty string  $\epsilon$ .

Superficially, it would appear that deterministic and non-deterministic finite state automata are entirely separate beasts. It turns out, however, that they are equivalent. For any language recognized by an NFA, there exists a DFA that recognizes that language and vice versa. The algorithm to make the conversion from NFA to DFA is relatively simple, even if the resulting DFA is considerably more complex than the original NFA. After the jump I will



prove this equivalence and also step through a short example of converting an NFA to an equivalent DFA.

## NFA and DFA Equivalence Theorem Proof

Before continuing, let's formally state the theorem we are proving:

### Theorem

Let language  $L \subseteq \Sigma^*$ , and suppose  $L$  is accepted by NFA  $N = (\Sigma, Q, q_0, F, \delta)$ . There exists a DFA  $D = (\Sigma, Q', q'_0, F', \delta')$  that also accepts  $L$ . ( $L(N) = L(D)$ ).

By allowing each state in the DFA  $D$  to represent a set of states in the NFA  $N$ , we are able to prove through induction that  $D$  is equivalent to  $N$ . Before we begin the proof, let's define the parameters of  $D$ :

- $Q'$  is equal to the power set of  $Q$ ,  $Q' = 2^Q$
- $q'_0 = \{q_0\}$
- $F'$  is the set of states in  $Q'$  that contain any element of  $F$ ,  $F' = \{q \in Q' \mid q \cap F \neq \emptyset\}$
- $\delta'$  is the transition function for  $D$ .  $\delta'(q, a) = \bigcup_{p \in q} \delta(p, a)$  for  $q \in Q'$  and  $a \in \Sigma$ .

Remember that each state in the set of states  $Q'$  in  $D$  is a set of states itself from  $Q$  in  $N$ . For each state  $p$  in state  $q$  in  $Q'$  of  $D$  ( $p$  is a single state from  $Q$ ), determine the transition  $\delta(p, a)$ .  $\delta(q, a)$  is the union of all  $\delta(p, a)$ .

Now we will prove that  $\widehat{\delta'}(q'_0, x) = \widehat{\delta}(q_0, x)$  for every  $x$ . i.e.,  $L(D) = L(N)$

### Basis Step

Let  $x$  be the empty string  $\epsilon$ .

$$\begin{aligned} \widehat{\delta'}(q'_0, x) &= \widehat{\delta'}(q'_0, \epsilon) \\ &= q'_0 \\ &= \{q_0\} \\ &= \widehat{\delta}(q_0, \epsilon) \\ &= \widehat{\delta}(q_0, x) \end{aligned}$$

### Inductive Step

Assume that for any  $y$  with  $|y| \geq 0$ ,  $\widehat{\delta'}(q'_0, y) = \widehat{\delta}(q_0, y)$ .

If we let  $n = |y|$ , then we need to prove that for a string  $z$  with

$$|z| = n + 1, \widehat{\delta'}(q'_0, z) = \widehat{\delta}(q_0, z).$$

We can represent the string  $z$  as a concatenation of string  $y$  ( $|y| = n$ ) and symbol  $a$ , from the alphabet  $\Sigma$  ( $a \in \Sigma$ ). So,  $z = y_a$ .

$$\widehat{\delta'}(q'_0, z) = \widehat{\delta'}(q'_0, y_a)$$

$$\begin{aligned}
&= \delta'(\hat{\delta}(q'_0, y), a) \\
&= \delta'(\hat{\delta}(q_0, y), a) \quad (\text{by assumption}) \\
&= \bigcup_{p \in \hat{\delta}(q_0, y)} \hat{\delta}(p, a) \quad (\text{by definition of } \delta') \\
&= \hat{\delta}(q_0, ay) \\
&= \hat{\delta}(q_0, z)
\end{aligned}$$

DFA D accepts a string  $x$  iff  $\hat{\delta}(q'_0, x) \in F'$ . From the above it follows that D accepts  $x$  iff  $\hat{\delta}(q'_0, x) \cap F \neq \emptyset$ .

So a string is accepted by DFA D if, and only if, it is accepted by NFA N.

### NFA to DFA Conversion Example

From the proof, we can tease out an algorithm that will allow us to convert any non-deterministic finite state automaton (NFA) to an equivalent deterministic finite state automaton (DFA). That is, the language accepted by the DFA is identical that accepted by the NFA.

#### Algorithm

Given NFA  $N = (\Sigma, Q, q_0, F, \delta)$  we want to build DFA  $D = (\Sigma, Q', q'_0, F', \delta')$ . Here's how:

- Initially  $Q' = \emptyset$ .
- Add  $q_0$  to  $Q'$ .
- For each state in  $Q$  find the set of possible states for each input symbol using  $N$ 's transition table,  $\delta$ . Add this set of states to  $Q'$ , if it is not already there.
- The set of final states of  $D$ ,  $F'$ , will be all of the states in  $Q'$  that contain in them a state that is in  $F$ .

Working through an example may aid in understanding these steps.

Consider the following NFA  $N = (\Sigma, Q, q_0, F, \delta)$

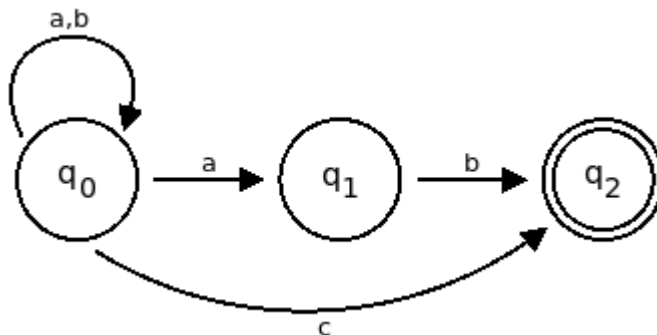


Figure 2.9: NFA

$$\Sigma = \{a, b, c\}$$

$$Q = \{q_0, q_1, q_2\}$$

$$F = \{q_2\}$$

We wish to construct DFA  $D = (\Sigma, Q', q'_0, F', \delta')$ . Following the steps in the conversion algorithm:

$$Q' = \emptyset$$

$$Q' = \{q_0\}$$

For every state in  $Q$ , find the set of states for each input symbol using  $N$ 's transition table,  $\delta$ . If the set is not  $Q$ , add it to  $Q'$ . We can start building the transition table  $\delta'$  for  $D$  by first examining  $q_0$ .

State	A	B	c
$q_0$	$\{q_0, q_1\}$	$q_0$	$q_2$

**Table 3**

$q_0$  is already in  $Q'$  so we don't add it.  $\{q_0, q_1\}$  is considered a single state, so we add it and  $q_2$  to  $Q'$ .

$$Q' = \{q_0, \{q_0, q_1\}, q_2\}$$

$\delta'$  now looks like:

State	A	b	c
$q_0$	$\{q_0, q_1\}$	$q_0$	$q_2$
$\{q_0, q_1\}$	?	?	?
$q_2$	?	?	?

**Table 4**

To fill in the transitions for  $\{q_0, q_1\}$ , you need to determine per symbol how each individual state in the set transitions and take the union of these states.

$$\delta'(\{q_0, q_1\}, a) = \delta(q_0, a) \cup \delta(q_1, a) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$$

$$\delta'(\{q_0, q_1\}, b) = \delta(q_0, b) \cup \delta(q_1, b) = c \cup \{q_2\} = \{q_0, q_2\}$$

$$\delta'(\{q_0, q_1\}, c) = \delta(q_0, c) \cup \delta(q_1, c) = \{q_2\} \cup \emptyset = \{q_2\}$$

The only new state here is  $\{q_0, q_2\}$ . We add it to  $Q'$ :

$$Q' = \{q_0, \{q_0, q_1\}, q_2, \{q_0, q_2\}\}$$

And update the transition table  $\delta'$ :

State	A	b	c
<b>q<sub>0</sub></b>	{q <sub>0</sub> , q <sub>1</sub> }	q <sub>0</sub>	q <sub>2</sub>
<b>{q<sub>0</sub>, q<sub>1</sub>}</b>	{q <sub>0</sub> , q <sub>1</sub> }	{q <sub>0</sub> , q <sub>2</sub> }	q <sub>2</sub>
<b>q<sub>2</sub></b>	?	?	?
<b>{q<sub>0</sub>, q<sub>2</sub>}</b>	?	?	?

**Table5**

q<sub>2</sub> has no outgoing transitions so the transition table is simple to update.

State	A	b	C
<b>q<sub>0</sub></b>	{q <sub>0</sub> , q <sub>1</sub> }	q <sub>0</sub>	q <sub>2</sub>
<b>{q<sub>0</sub>, q<sub>1</sub>}</b>	{q <sub>0</sub> , q <sub>1</sub> }	{q <sub>0</sub> , q <sub>2</sub> }	q <sub>2</sub>
<b>q<sub>2</sub></b>	-	-	-
<b>{q<sub>0</sub>, q<sub>2</sub>}</b>	?	?	?

**Table 6**

Now calculate the transitions for {q<sub>0</sub>, q<sub>2</sub>}:

$$\delta'(\{q_0, q_2\}, a) = \delta(q_0, a) \cup \delta(q_2, a) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$$

$$\delta'(\{q_0, q_2\}, b) = \delta(q_0, b) \cup \delta(q_2, b) = \{q_0\} \cup \emptyset = \{q_0\}$$

$$\delta'(\{q_0, q_2\}, c) = \delta(q_0, c) \cup \delta(q_2, c) = \{q_2\} \cup \emptyset = \{q_2\}$$

There are no new states to add to Q'. The updated transition table looks like:

State	A	b	C
<b>q<sub>0</sub></b>	{q <sub>0</sub> , q <sub>1</sub> }	q <sub>0</sub>	q <sub>2</sub>
<b>{q<sub>0</sub>, q<sub>1</sub>}</b>	{q <sub>0</sub> , q <sub>1</sub> }	{q <sub>0</sub> , q <sub>2</sub> }	q <sub>2</sub>

$q_2$	-	-	-
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$q_0$	$q_2$

**Table 7**

Since we've inspected all of the states in  $Q$  and no longer have any states to add to  $Q'$ , we are finished.  $D$ 's set of final states  $F'$  are those states that include states in  $F$ . So,  $F' = \{q_2, \{q_0, q_2\}\}$ .

For our completed DFA  $D = (\Sigma, Q', q'_0, F', \delta')$ :

$$\Sigma = \{a, b\}$$

$$Q' = \{q_0, \{q_0, q_1\}, q_2, \{q_0, q_2\}\}$$

$$q'_0 = q_0$$

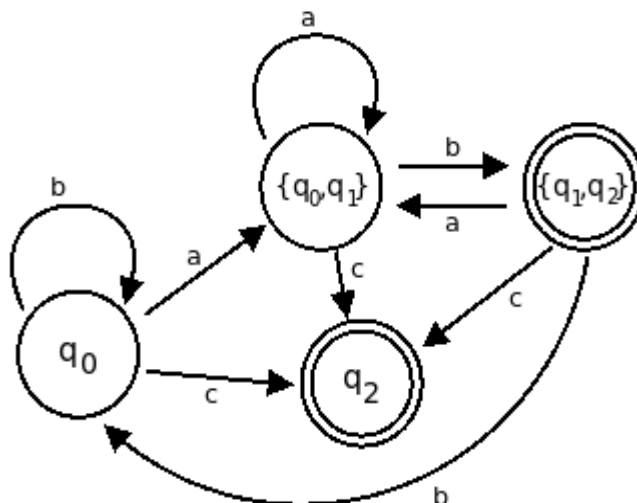
$$F' = \{q_2, \{q_0, q_2\}\}$$

$\delta'$  is:

State	A	b	C
$q_0$	$\{q_0, q_1\}$	$q_0$	$q_2$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$q_2$
$q_2$	-	-	-
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$q_0$	$q_2$

**Table 8**

The transition graph for this DFA looks like:



B

**Figure: 2.10: transition graph for DFA**

## 2.5 Finite automata with epsilon transitions

Non-deterministic finite automata(NFA) is a finite automaton where for some cases when a specific input is given to the current state, the machine goes to multiple states or more than 1 states. It can contain  $\epsilon$  move. It can be represented as  $M = \{Q, \Sigma, \delta, q_0, F\}$ .

Where

- $Q$ : finite set of states
- $\Sigma$ : finite set of the input symbol
- $q_0$ : initial state
- $F$ : final state
- $\delta$ : Transition function

**NFA with  $\epsilon$  move:** If any FA contains  $\epsilon$  transaction or move, the finite automata is called NFA with  $\epsilon$  move.

**$\epsilon$ -closure:**  $\epsilon$ -closure for a given state  $A$  means a set of states which can be reached from the state  $A$  with only  $\epsilon$ (null) move including the state  $A$  itself.

### Steps for converting NFA with $\epsilon$ to DFA:

Step 1: We will take the  $\epsilon$ -closure for the starting state of NFA as a starting state of DFA.

Step 2: Find the states for each input symbol that can be traversed from the present. That means the union of transition value and their closures for each state of NFA present in the current state of DFA.

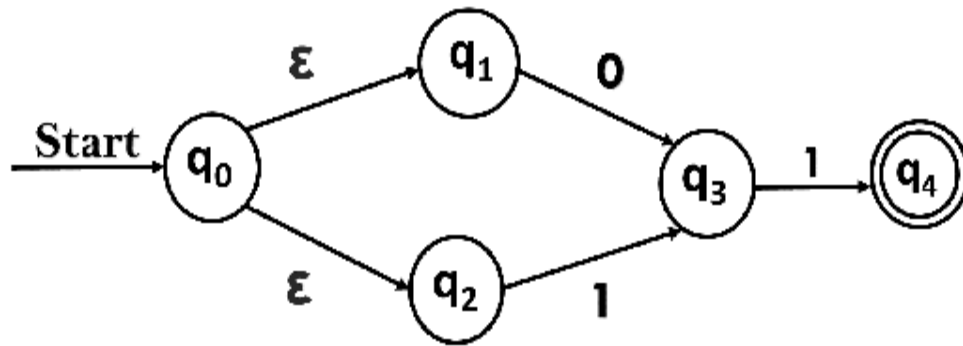
Step 3: If we found a new state, take it as current state and repeat step 2.

Step 4: Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.

Step 5: Mark the states of DFA as a final state which contains the final state of NFA.

### Example:

Convert the NFA with  $\epsilon$  into its equivalent DFA.



**Figure: 2.11: transition graph for NFA with  $\epsilon$  into its equivalent DFA**

**Solution:**

Let us obtain  $\epsilon$ -closure of each state.

$$\epsilon\text{-closure } \{q_0\} = \{q_0, q_1, q_2\}$$

$$\epsilon\text{-closure } \{q_1\} = \{q_1\}$$

$$\epsilon\text{-closure } \{q_2\} = \{q_2\}$$

$$\epsilon\text{-closure } \{q_3\} = \{q_3\}$$

$$\epsilon\text{-closure } \{q_4\} = \{q_4\}$$

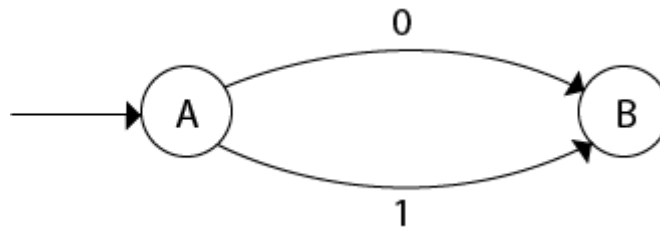
Now, let  $\epsilon\text{-closure } \{q_0\} = \{q_0, q_1, q_2\}$  be state A.

Hence

$$\begin{aligned}
 \delta'(A, 0) &= \epsilon\text{-closure } \{\delta((q_0, q_1, q_2), 0)\} \\
 &= \epsilon\text{-closure } \{\delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)\} \\
 &= \epsilon\text{-closure } \{q_3\} \\
 &= \{q_3\} \quad \text{call it as state B.}
 \end{aligned}$$

$$\begin{aligned}
 \delta'(A, 1) &= \epsilon\text{-closure } \{\delta((q_0, q_1, q_2), 1)\} \\
 &= \epsilon\text{-closure } \{\delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)\} \\
 &= \epsilon\text{-closure } \{q_3\} \\
 &= \{q_3\} = B.
 \end{aligned}$$

The partial DFA will be



**Figure 2.12: Partial DFA**

Now,

$$\delta'(B, 0) = \varepsilon\text{-closure } \{\delta(q_3, 0)\}$$

$$= \phi$$

$$\delta'(B, 1) = \varepsilon\text{-closure } \{\delta(q_3, 1)\}$$

$$= \varepsilon\text{-closure } \{q_4\}$$

$$= \{q_4\} \quad \text{i.e. state C}$$

For state C:

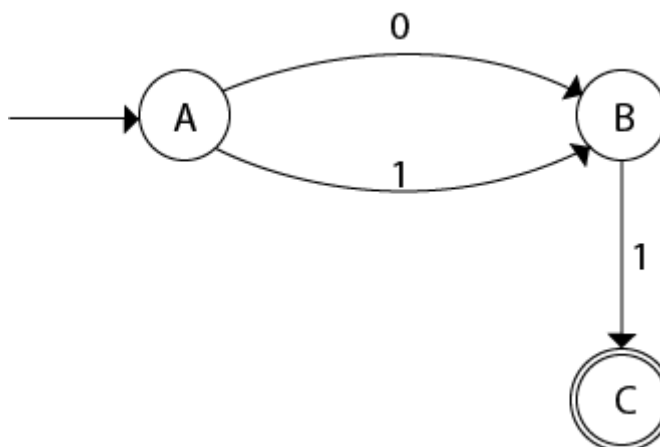
$$\delta'(C, 0) = \varepsilon\text{-closure } \{\delta(q_4, 0)\}$$

$$= \phi$$

$$\delta'(C, 1) = \varepsilon\text{-closure } \{\delta(q_4, 1)\}$$

$$= \phi$$

The DFA will be,



**Figure 2.13: DFA**

## 2.6 Removal of epsilon transitions

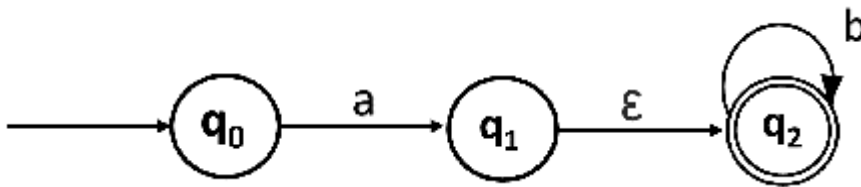


NFA with  $\epsilon$  can be converted to NFA without  $\epsilon$ , and this NFA without  $\epsilon$  can be converted to DFA. To do this, we will use a method, which can remove all the  $\epsilon$  transition from given NFA. The method will be:

1. Find out all the  $\epsilon$  transitions from each state from Q. That will be called as  $\epsilon$ -closure  $\{q_i\}$  where  $q_i \in Q$ .
2. Then  $\delta'$  transitions can be obtained. The  $\delta'$  transitions mean a  $\epsilon$ -closure on  $\delta$  moves.
3. Repeat Step-2 for each input symbol and each state of given NFA.
4. Using the resultant states, the transition table for equivalent NFA without  $\epsilon$  can be built.

**Example:**

Convert the following NFA with  $\epsilon$  to NFA without  $\epsilon$ .



**Figure 2.14: NFA with  $\epsilon$**

**Solutions:** We will first obtain  $\epsilon$ -closures of  $q_0$ ,  $q_1$  and  $q_2$  as follows:

$$\epsilon\text{-closure}(q_0) = \{q_0\}$$

$$\epsilon\text{-closure}(q_1) = \{q_1, q_2\}$$

$$\epsilon\text{-closure}(q_2) = \{q_2\}$$

Now the  $\delta'$  transition on each input symbol is obtained as:

$$\begin{aligned}
 \delta'(q_0, a) &= \epsilon\text{-closure}(\delta(\delta^{\wedge}(q_0, \epsilon), a)) \\
 &= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0), a)) \\
 &= \epsilon\text{-closure}(\delta(q_0, a)) \\
 &= \epsilon\text{-closure}(q_1) \\
 &= \{q_1, q_2\}
 \end{aligned}$$

$$\delta'(q_0, b) = \epsilon\text{-closure}(\delta(\delta^{\wedge}(q_0, \epsilon), b))$$

$$\begin{aligned}
&= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_0), b)) \\
&= \varepsilon\text{-closure}(\delta(q_0, b)) \\
&= \Phi
\end{aligned}$$

Now the  $\delta'$  transition on  $q_1$  is obtained as:

$$\begin{aligned}
\delta'(q_1, a) &= \varepsilon\text{-closure}(\delta(\delta^\wedge(q_1, \varepsilon), a)) \\
&= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_1), a)) \\
&= \varepsilon\text{-closure}(\delta(q_1, q_2), a) \\
&= \varepsilon\text{-closure}(\delta(q_1, a) \cup \delta(q_2, a)) \\
&= \varepsilon\text{-closure}(\Phi \cup \Phi) \\
&= \Phi
\end{aligned}$$

$$\begin{aligned}
\delta'(q_1, b) &= \varepsilon\text{-closure}(\delta(\delta^\wedge(q_1, \varepsilon), b)) \\
&= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_1), b)) \\
&= \varepsilon\text{-closure}(\delta(q_1, q_2), b) \\
&= \varepsilon\text{-closure}(\delta(q_1, b) \cup \delta(q_2, b)) \\
&= \varepsilon\text{-closure}(\Phi \cup q_2) \\
&= \{q_2\}
\end{aligned}$$

The  $\delta'$  transition on  $q_2$  is obtained as:

$$\begin{aligned}
\delta'(q_2, a) &= \varepsilon\text{-closure}(\delta(\delta^\wedge(q_2, \varepsilon), a)) \\
&= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_2), a)) \\
&= \varepsilon\text{-closure}(\delta(q_2, a)) \\
&= \varepsilon\text{-closure}(\Phi) \\
&= \Phi
\end{aligned}$$

$$\begin{aligned}
\delta'(q_2, b) &= \varepsilon\text{-closure}(\delta(\delta^\wedge(q_2, \varepsilon), b)) \\
&= \varepsilon\text{-closure}(\delta(\varepsilon\text{-closure}(q_2), b)) \\
&= \varepsilon\text{-closure}(\delta(q_2, b)) \\
&= \varepsilon\text{-closure}(q_2)
\end{aligned}$$

$$= \{q_2\}$$

Now we will summarize all the computed  $\delta'$  transitions:

$$\delta'(q_0, a) = \{q_0, q_1\}$$

$$\delta'(q_0, b) = \Phi$$

$$\delta'(q_1, a) = \Phi$$

$$\delta'(q_1, b) = \{q_2\}$$

$$\delta'(q_2, a) = \Phi$$

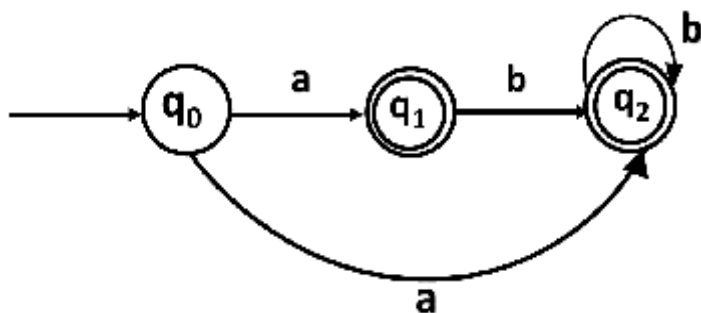
$$\delta'(q_2, b) = \{q_2\}$$

The transition table can be:

State	a	b
<b>q<sub>0</sub></b>	{q <sub>1</sub> , q <sub>2</sub> }	$\phi$
<b>* q<sub>1</sub></b>	$\phi$	{q <sub>2</sub> }
<b>* q<sub>2</sub></b>	$\phi$	{q <sub>2</sub> }

**Table 9**

State  $q_1$  and  $q_2$  become the final state as  $\epsilon$ -closure of  $q_1$  and  $q_2$  contain the final state  $q_2$ . The NFA can be shown by the following transition diagram:



**Figure 2.15: Transition diagram of NFA**

---

## 2.7 Summary

---

In this unit you have learnt about Finite State Systems, Basic Definitions Non-Deterministic finite automata (N DFA), Deterministic finite automata (DFA), Equivalence of DFA and N DFA, Finite automata with epsilon transitions, Removal of epsilon transitions.

- Finite Automata (FA) is the simplest machine to recognize patterns. The finite automata or finite state machine is an abstract machine which have five elements or tuple.
- It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol.
- In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.
- In N DFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-Deterministic Automaton. As it has finite number of states, the machine is called Non-Deterministic Finite Machine or Non-Deterministic Finite Automaton.

---

## 2.8 Review Questions

---

Q1. How do you create a DFA in automata? Explain with example

Q2. Can NFA have multiple final states? Elaborate your answer.

Q3. Design FA with  $\Sigma = \{0, 1\}$  accepts the set of all strings with three consecutive 0's.

Q4. Design a DFA  $L(M) = \{w \mid w \in \{0, 1\}^*\}$  and  $W$  is a string that does not contain consecutive 1's

Q5. Design an NFA in which all the string contains a substring 1110.

Q6. Under which of the following operation, NFA is not closed?

- a) Negation
- b) Kleene
- c) Concatenation
- d) None of the mentioned

Q7. Which of the following is an application of Finite Automaton?

- a) Compiler Design
- b) Grammar Parsers
- c) Text Search
- d) All of the mentioned

---

## **UNIT-3 Introduction to Machines**

---

### **Structure**

3.0 Introduction

3.1 Concept of basic Machine

3.2 Properties and limitations of FSM

3.3 Moore and mealy Machines

3.4 Equivalence of Moore and Mealy machines.

3.5 Minimization of DFA.

3.6 Summary

3.7 Review Questions

## 3.0 Introduction

This is the third and last unit of this block. In this unit, there are seven sections. First section explains about the basic concept of machine. Second section that is Section 3.2 define properties and limitations of FSM. Section 3.3 provide the detail knowledge of Moore and mealy Machines; Equivalence of Moore and Mealy machines describe in the section 3.4. You will learn about Minimization of DFA in the Section 3.5. Last two sections i.e. Section 3.6 and 3.7 provide summary and review questions of the unit respectively.

### Objectives

After studying this unit, you should be able to:

- Basic Concept of Machine
- Properties and limitations of FSM.
- Moore and mealy Machines
- Equivalence of Moore and Mealy machines
- Minimization of DFA.

## 3.1 Concept of basic Machine

Automata Theory is an exciting, theoretical branch of computer science. It established its roots during the 20th Century, as mathematicians began developing - both theoretically and literally - machines which imitated certain features of man, completing calculations more quickly and reliably. The word automaton itself, closely related to the word "automation", denotes automatic processes carrying out the production of specific processes. Simply stated, automata theory deals with the logic of computation with respect to simple machines, referred to as automata. Through automata, computer scientists are able to understand how machines compute functions and solve problems and more importantly, what it means for a function to be defined as computable or for a question to be described as decidable.

Automatons are abstract models of machines that perform computations on an input by moving through a series of states or configurations. At each state of the computation, a transition function determines the next configuration on the basis of a finite portion of the present configuration. As a result, once the computation reaches an accepting configuration, it accepts that input. The most general and powerful automata is the Turing machine.

The major objective of automata theory is to develop methods by which computer scientists can describe and analyse the dynamic behaviour of discrete systems, in which signals are sampled periodically. The behaviour of these discrete systems is determined by the way that the system is constructed from storage and combinational elements. Characteristics of such machines include:

- Inputs: assumed to be sequences of symbols selected from a finite set  $I$  of input signals. Namely, set  $I$  is the set  $\{x_1, x_2, x_3, \dots, x_k\}$  where  $k$  is the number of inputs.
- Outputs: sequences of symbols selected from a finite set  $Z$ . Namely, set  $Z$  is the set  $\{y_1, y_2, y_3, \dots, y_m\}$  where  $m$  is the number of outputs.
- States: finite set  $Q$ , whose definition depends on the type of automaton.

There are four major families of automaton:

- Finite-state machine
- Pushdown automata
- Linear-bounded automata
- Turing machine

The families of automata above can be interpreted in a hierarchical form, where the finite-state machine is the simplest automata and the Turing machine is the most complex. The focus of this project is on the finite-state machine and the Turing machine. A Turing machine is a finite-state machine yet the inverse is not true.

## Finite State Machines

The exciting history of how finite automata became a branch of computer science illustrates its wide range of applications. The first people to consider the concept of a finite-state machine included a team of biologists, psychologists, mathematicians, engineers and some of the first computer scientists. They all shared a common interest: to model the human thought process, whether in the brain or in a computer. Warren McCulloch and Walter Pitts, two neurophysiologists, were the first to present a description of finite automata in 1943. Their paper, entitled, "A Logical Calculus Immanent in Nervous Activity", made significant contributions to the study of neural network theory, theory of automata, the theory of computation and cybernetics. Later, two computer scientists, G.H. Mealy and E.F. Moore, generalized the theory to much more powerful machines in separate papers, published in 1955-56. The finite-state machines, the Mealy machine and the Moore machine, are named in recognition of their work. While the Mealy machine determines its outputs through the current state and the input, the Moore machine's output is based upon the current state alone.

An automaton in which the state set  $Q$  contains only a finite number of elements is called a finite-state machine (FSM). FSMs are abstract machines, consisting of a set of states (set  $Q$ ), set of input events (set  $I$ ), a set of output events (set  $Z$ ) and a state transition function. The state transition function takes the current state and an input event and returns the new set of output events and the next state. Therefore, it can be seen as a Function, which maps an ordered sequence of input events into a corresponding sequence, or set, of output events.



State transition function:  $I \rightarrow Z$

Finite-state machines are ideal computation models for a small amount of memory, and do not maintain memory. This mathematical model of a machine can only reach a finite number of states and transitions between these states. Its main application is in mathematical problem analysis. Finite-machines are also used for purposes aside from general computations, such as to recognize regular languages.

In order to fully understand conceptually a finite-state machine, consider an analogy to an elevator:

An elevator is a mechanism that does not remember all previous requests for service but the current floor, the direction of motion (up or down) and the collection of not-yet satisfied requests for services. Therefore, at any given moment of time, an elevator in operation would be defined by the following mathematical terms:

- States: finite set of states to reflect the history of the customer's requests.
- Inputs: finite set of input, depending on the number of floors the elevator is able to access. We can use the set  $I$ , whose size is the number of floors in the building.
- Outputs: finite set of output, depending on the need for the elevator to go up or down, according to customers' needs.

A finite-state machine is formally defined as a 5-tuple  $(Q, I, Z, \partial, W)$  such that:

- $Q$  = finite set of states
- $I$  = finite set of input symbols
- $Z$  = finite set of output symbols
- $\partial$  = mapping of  $I \times Q$  into  $Q$  called the state transition function, i.e.  $I \times Q \rightarrow Q$
- $W$  = mapping  $W$  of  $I \times Q$  onto  $Z$ , called the output function
- $A$  = set of accept states where  $F$  is a subset of  $Q$

From the mathematical interpretation above, it can be said that a finite-state machine contains a finite number of states. Each state accepts a finite number of inputs, and each state has rules that describe the action of the machine for every input, represented in the state transition mapping function. At the same time, an input may cause the machine to change states. For every input symbol, there is exactly one transition out of each state. In addition, we can say that any 5-tuple set which is accepted by nondeterministic finite automata can also be accepted by deterministic finite automata.

When considering finite-state machines, it is important to keep in mind that the mechanical process inside the automata that leads to the calculation of outputs and change of states is not emphasized or delved into detail; it is instead considered a "black box", as illustrated below:

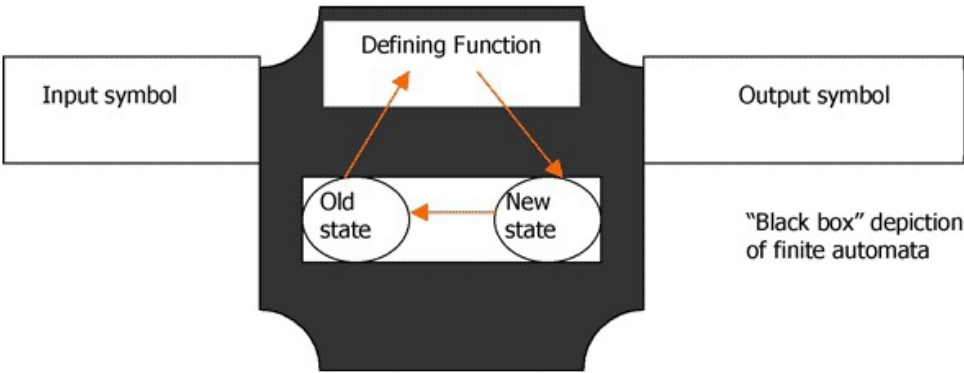


Figure 3.1: Black Box

Having finite, constant amounts of memory, the internal states of an FSM carry no further structure. They can easily be represented using state diagrams, as seen below:

State	Input	Output	New State
q <sub>0</sub>	X	0	q <sub>0</sub>
q <sub>0</sub>	a	1	q <sub>1</sub>
q <sub>0</sub>	b	1	q <sub>1</sub>
q <sub>1</sub>	X	0	q <sub>0</sub>
q <sub>1</sub>	a	0	q <sub>1</sub>
q <sub>1</sub>	b	0	q <sub>1</sub>

Table 10

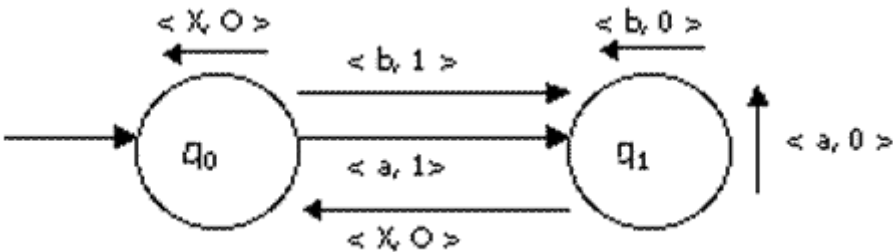


Figure 3.2: State diagram of FSM

The state diagram illustrates the operation of an automaton. States are represented by nodes of graphs, transitions by the arrows or branches, and the corresponding inputs and outputs are denoted by symbols. The arrow entering from the left into q<sub>0</sub> shows that q<sub>0</sub> is the initial state of the machine. Moves

that do not involve changes of states are indicated by arrows along the sides of individual nodes. These arrows are known as self-loops.

There exist several types of finite-state machines, which can be divided into three main categories:

- acceptors: either accept the input or do not
- recognizers: either recognize the input or do not
- transducers: generate output from given input

Applications of finite-state machines are found in a variety of subjects. They can operate on languages with a finite number of words (standard case), an infinite number of words (Rabin automata, B rche automata), various types of trees, and in hardware circuits, where the input, the state and the output are bit vectors of a fixed size.

### **Finite State vs. Turing Machines**

The simplest automata used for computation is a finite automaton. It can compute only very primitive functions; therefore, it is not an adequate computation model. In addition, a finite-state machine's inability to generalize computations hinders its power.

The following is an example to illustrate the difference between a finite-state machine and a Turing machine:

Imagine a Modern CPU. Every bit in a machine can only be in two states (0 or 1). Therefore, there are a finite number of possible states. In addition, when considering the parts of a computer a CPU interacts with, there are a finite number of possible inputs from the computer's mouse, keyboard, hard disk, different slot cards, etc. As a result, one can conclude that a CPU can be modelled as a finite-state machine.

Now, consider a computer. Although every bit in a machine can only be in two different states (0 or 1), there are an infinite number of interactions within the computer as a whole. It becomes exceeding difficult to model the workings of a computer within the constraints of a finite-state machine. However, higher-level, infinite and more powerful automata would be capable of carrying out this task.

World-renowned computer scientist Alan Turing conceived the first "infinite" (or unbounded) model of computation: The Turing machine, in 1936, to solve the Entscheidungs problem. The Turing machine can be thought of as a finite automaton or control unit equipped with an infinite storage (memory). Its "memory" consists of an infinite number of one-dimensional array of cells. Turing's machine is essentially an abstract model of modern-day computer execution and storage, developed in order to provide a precise mathematical definition of an algorithm or mechanical procedure.

While an automaton is called finite if its model consists of a finite number of states and functions with finite strings of input and output, infinite automata have an "accessory" - either a stack or a tape that can be moved to the right or left, and can meet the same demands made on a machine.

A Turing machine is formally defined by the set  $[Q, \Sigma, \Gamma, \delta, q_0, B, F]$  where

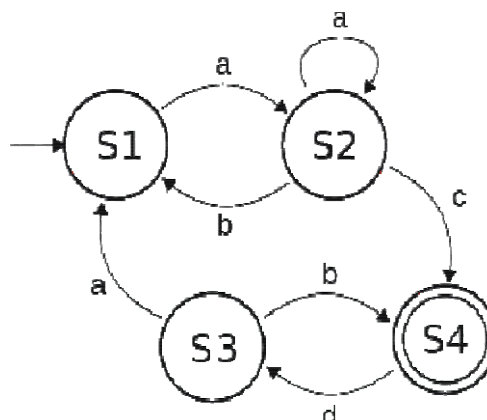
- $Q$  = finite set of states, of which one state  $q_0$  is the initial state
- $\Sigma$  = a subset of  $\Gamma$  not including  $B$ , is the set of input symbols
- $\Gamma$  = finite set of allowable tape symbols
- $\delta$  = the next move function, a mapping function from  $Q \times \Gamma$  to  $Q \times \Gamma \times \{L, R\}$ , where  $L$  and  $R$  denote the directions left and right respectively
- $q_0$  = in set  $Q$  as the start state
- $B$  = a symbol of  $\Gamma$ , as the blank
- $F \subseteq Q$  the set of final states

Therefore, the major difference between a Turing machine and two-way finite automata (FSM) lies in the fact that the Turing machine is capable of changing symbols on its tape and simulating computer execution and storage. For this reason, it can be said that the Turing Machine has the power to model all computations that can be calculated today through modern computers.

### 3.2 Properties and limitations of FSM

FSMs are most commonly represented by state diagrams, which are also called state transition diagrams. The state diagram is basically a directed graph where each vertex represents a state and each edge represents a transition between two states.

Another common representation of FSMs is state transition tables. In these tables, every column corresponds to a state, every row corresponds to an event category. Values in the table cells give the states resulting from the respective transitions. Table cells also can be used for specifying actions related to transitions.



FSMs are used in games; they are most recognized for being utilized in artificial intelligence, and however, they are also frequent in executions of navigating parsing text, input handling of the customer, as well as network protocols.

These are restricted in computational power; they have the good quality of being comparatively simple to recognize. So, they are frequently used by software developers as well as system designers for summarizing the performance of a difficult system.

The finite state machines are applicable in vending machines, video games, traffic lights, controllers in CPU, text parsing, analysis of protocol, recognition of speech, language processing, etc.

### **Advantages of Finite State Machine**

The advantages of Finite State Machine include the following.

- Finite state machines are flexible
- Easy to move from a significant abstract to a code execution
- Low processor overhead
- Easy determination of reachability of a state

### **Limitations**

The disadvantages of the finite state machine include the following

- The expected character of deterministic finite state machines can be not needed in some areas like computer games
- The implementation of huge systems using FSM is hard for managing without any idea of design.
- Not applicable for all domains
- The orders of state conversions are inflexible.

---

## **Check your progress**

---

Q1. What is the role of Finite state machine in theory of automata?

Q2. Define all the properties and limitations of automata.

### 3.3 Moore and mealy Machines

Finite automata may have outputs corresponding to each transition. There are two types of finite state machines that generate output –

- Mealy Machine
- Moore machine

#### Mealy Machine

A Mealy Machine is an FSM whose output depends on the present state as well as the present input.

It can be described by a 6 tuple  $(Q, \Sigma, O, \delta, X, q_0)$  where –

- $Q$  is a finite set of states.
- $\Sigma$  is a finite set of symbols called the input alphabet.
- $O$  is a finite set of symbols called the output alphabet.
- $\delta$  is the input transition function where  $\delta: Q \times \Sigma \rightarrow Q$
- $X$  is the output transition function where  $X: Q \times \Sigma \rightarrow O$
- $q_0$  is the initial state from where any input is processed ( $q_0 \in Q$ ).

Present state	Next state			
	input = 0		input = 1	
	State	Output	State	Output
→ a	b	$x_1$	c	$x_1$
b	b	$x_2$	d	$x_3$
c	d	$x_3$	c	$x_1$
d	d	$x_3$	d	$x_2$

Table 11

The state diagram of the above Mealy Machine is –

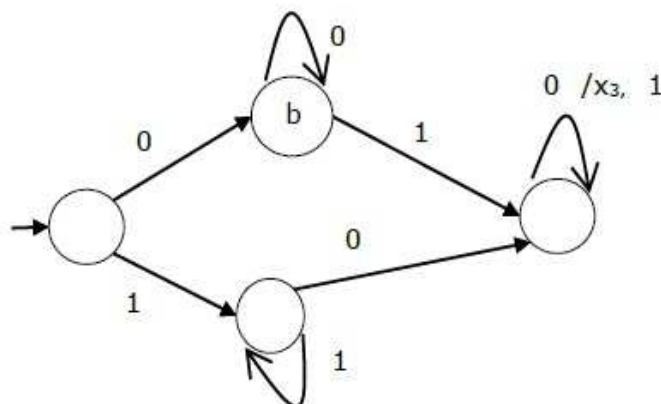


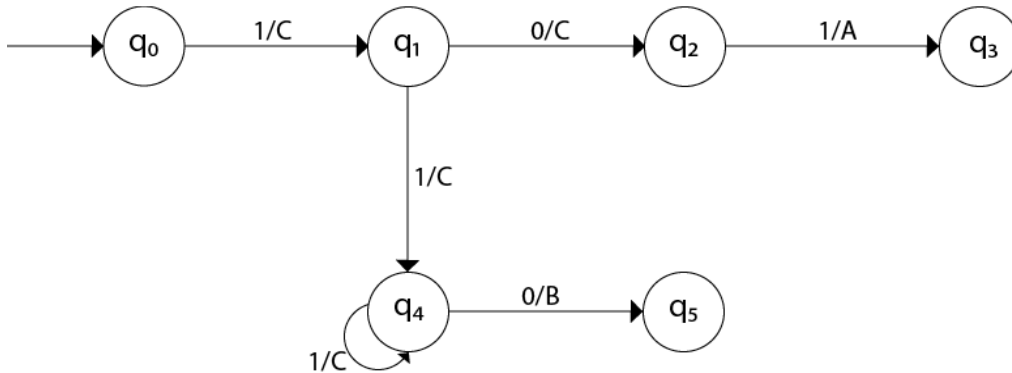
Figure 3.3: State diagram of Mealy Machine

### Example:

Design a Mealy machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.

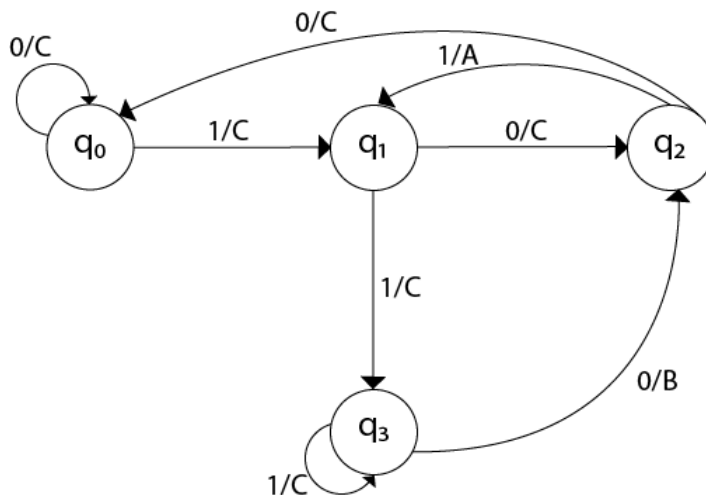
**Solution:** For designing such a machine, we will check two conditions, and those are 101 and 110. If we get 101, the output will be A. If we recognize 110, the output will be B. For other strings the output will be C.

The partial diagram will be:



**Figure 3.4 Partial diagram of Mealy Machine**

Now we will insert the possibilities of 0's and 1's for each state. Thus the Mealy machine becomes:



**Figure 3.5: Mealy Machine**

### Moore Machine

Moore machine is an FSM whose outputs depend on only the present state.

A Moore machine can be described by a 6 tuple  $(Q, \Sigma, O, \delta, X, q_0)$  where –

- $Q$  is a finite set of states.

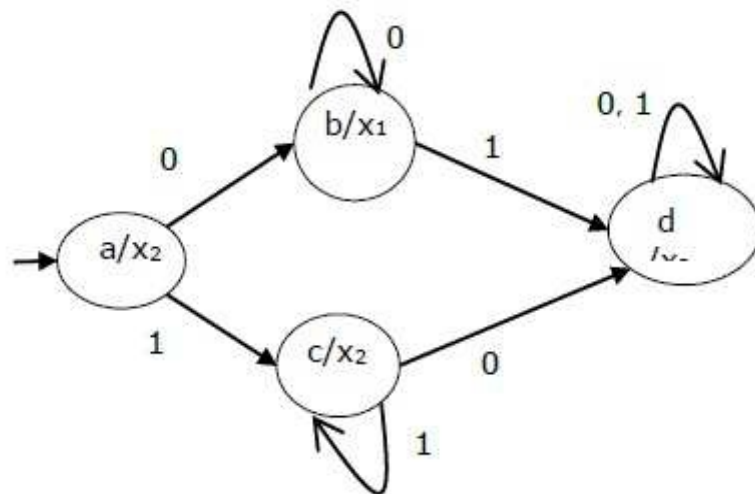
- $\Sigma$  is a finite set of symbols called the input alphabet.
- $O$  is a finite set of symbols called the output alphabet.
- $\delta$  is the input transition function where  $\delta: Q \times \Sigma \rightarrow Q$
- $X$  is the output transition function where  $X: Q \rightarrow O$
- $q_0$  is the initial state from where any input is processed ( $q_0 \in Q$ ).

The state table of a Moore Machine is shown below –

Present state	Next State		Output
	Input = 0	Input = 1	
→ a	b	c	$x_2$
b	b	d	$x_1$
c	c	d	$x_2$
d	d	d	$x_3$

**Table 12**

The state diagram of the above Moore Machine is –



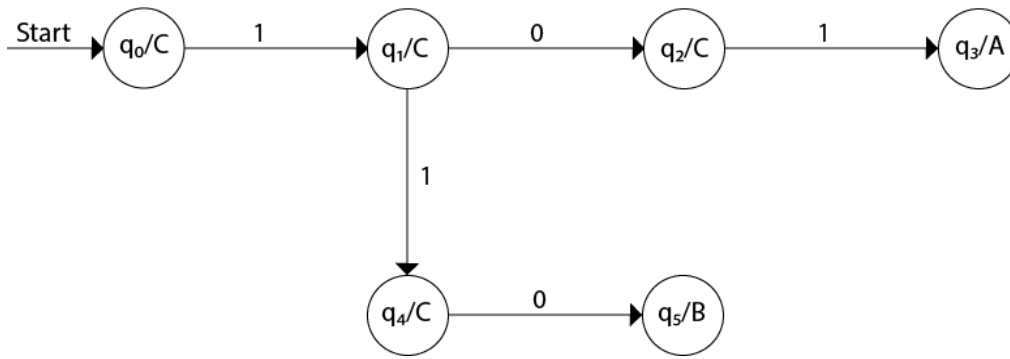
**Figure 3.6: The State diagram of Moore Machine**

### Example:

Design a Moore machine for a binary input sequence such that if it has a substring 101, the machine output A, if the input has substring 110, it outputs B otherwise it outputs C.

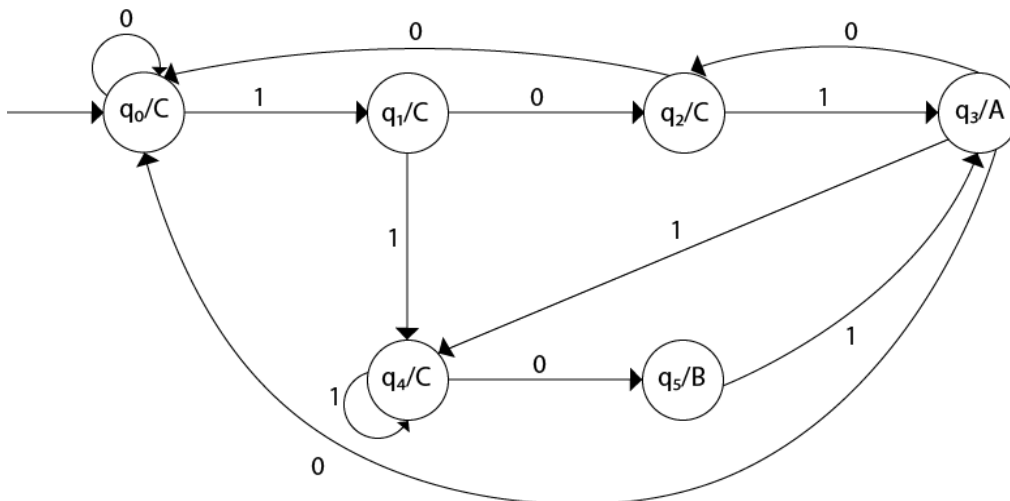
**Solution:** For designing such a machine, we will check two conditions, and those are 101 and 110. If we get 101, the output will be A, and if we recognize 110, the output will be B. For other strings, the output will be C.





**Figure 3.7: Partial diagram of Moore Machine**

Now we will insert the possibilities of 0's and 1's for each state. Thus the Moore machine becomes:



**Figure 3.8: Moore Machine Transition Diagram**

### Mealy Machine vs. Moore Machine

The following table highlights the points that differentiate a Mealy Machine from a Moore Machine.

Sr. No	Mealy Machine	Moore Machine
1	Output depends both upon the present state and the present input.	Output depends only upon the present state.
2	Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
3	The value of the output function is a function of the transitions	The value of the output function is a function of the current state

	and the changes, when the input logic on the present state is done.	and the changes at the clock edges, whenever state changes occur.
4	Mealy machines react faster to inputs. They generally react in the same clock cycle.	In Moore machines, more logic is required to decode the outputs resulting in more circuit delays. They generally react one clock cycle later.

**Table 13**

### 3.4 Equivalence of Moore and Mealy machines

Moore Machine to Mealy Machine

**Algorithm:**

Input – Moore Machine

Output – Mealy Machine

Step 1 – Take a blank Mealy Machine transition table format.

Step 2 – Copy all the Moore Machine transition states into this table format.

Step 3 – Check the present states and their corresponding outputs in the Moore Machine state table; if for a state  $Q_i$  output is  $m$ , copy it into the output columns of the Mealy Machine state table wherever  $Q_i$  appears in the next state.

**Example**

Let us consider the following Moore machine –

Present state	Next State		Output
	a = 0	b = 1	
→ a	d	b	1
b	a	d	0
c	c	c	0
d	b	a	1

**Table 14**

Now we apply above Algorithm to convert it to Mealy Machine.

**Step 1 & 2 –**

Present state	Next state			
	a = 0		b = 1	
	State	Output	State	Output
→ a	d		b	
b	a		d	
c	c		c	
d	b		a	

**Table 15****Step 3 –**

Present state	Next state			
	a = 0		b = 1	
	State	Output	State	Output
→ a	d	1	b	0
b	a	1	d	1
c	c	0	c	0
d	b	0	a	1

**Table 16****Mealy Machine to Moore Machine****Algorithm**

Input – Mealy Machine

Output – Moore Machine

Step 1 – Calculate the number of different outputs for each state ( $Q_i$ ) that are available in the state table of the Mealy machine.

Step 2 – If all the outputs of  $Q_i$  are same, copy state  $Q_i$ . If it has  $n$  distinct outputs, break  $Q_i$  into  $n$  states as  $Q_{in}$  where  $n = 0, 1, 2, \dots$

Step 3 – If the output of the initial state is 1, insert a new initial state at the beginning which gives 0 output.

**Example**

Let us consider the following Mealy Machine –

Present state	Next state			
	a = 0		b = 1	
	State	Output	State	Output
→ a	d	0	b	1
b	a	1	d	0
c	c	1	c	0
d	b	0	a	1

**Table 17**

Here, states 'a' and 'd' give only 1 and 0 outputs respectively, so we retain states 'a' and 'd'. But states 'b' and 'c' produce different outputs (1 and 0). So, we divide b into  $b_0$ ,  $b_1$  and c into  $c_0$ ,  $c_1$ .

Present state	Next State		Output
	a = 0	b = 1	
→ a	d	b	1
$b_0$	a	d	0
$b_1$	a	d	1

c <sub>0</sub>	c <sub>1</sub>	c <sub>0</sub>	0
c <sub>1</sub>	c <sub>1</sub>	c <sub>0</sub>	1
d	b <sub>0</sub>	a	0

**Table 18**

### 3.5 Minimization of DFA

Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM (finite state machine) with redundant states after minimizing the FSM.

We have to follow the various steps to minimize the DFA. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1.  $\delta(q, a) = p$
2.  $\delta(r, a) = p$

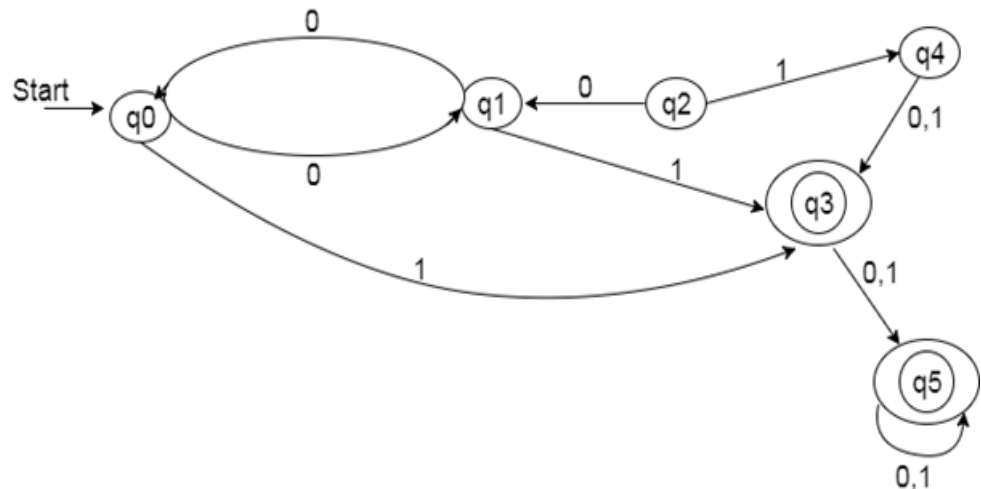
That means, find the two states which have the same value of a and b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

#### Example: Minimize the DFA



**Figure 3.9: Transition diagram of Minimization of DFA**

**Solution:**

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states.

State	0	1
→q <sub>0</sub>	q <sub>1</sub>	q <sub>3</sub>
q <sub>1</sub>	q <sub>0</sub>	q <sub>3</sub>
*q <sub>3</sub>	q <sub>5</sub>	q <sub>5</sub>
*q <sub>5</sub>	q <sub>5</sub>	q <sub>5</sub>

**Table 19**

Step 3: Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

State	0	1
→q <sub>0</sub>	q <sub>1</sub>	q <sub>3</sub>
q <sub>1</sub>	q <sub>0</sub>	q <sub>3</sub>

**Table 20**

2. Another set contains those rows, which starts from final states.

State	0	1
q <sub>3</sub>	q <sub>5</sub>	q <sub>5</sub>
q <sub>5</sub>	q <sub>5</sub>	q <sub>5</sub>

**Table 21**

Step 4: Set 1 has no similar rows so set 1 will be the same.

Step 5: In set 2, row 1 and row 2 are similar since q<sub>3</sub> and q<sub>5</sub> transit to the same state on 0 and 1. So skip q<sub>5</sub> and then replace q<sub>5</sub> by q<sub>3</sub> in the rest.

State	0	1
q <sub>3</sub>	q <sub>3</sub>	q <sub>3</sub>

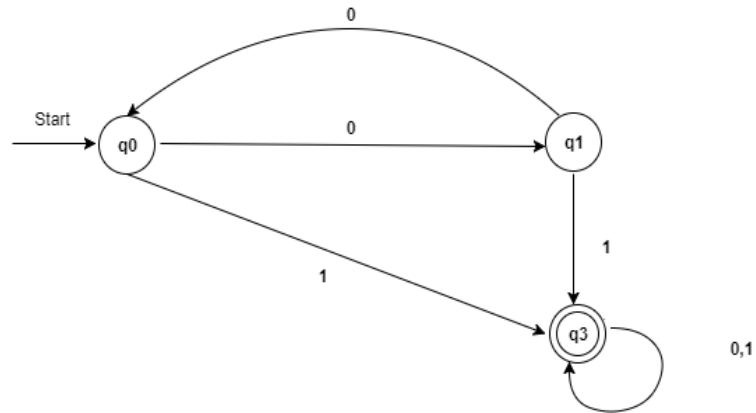
**Table 22**

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q <sub>0</sub>	q <sub>1</sub>	q <sub>3</sub>
q <sub>1</sub>	q <sub>0</sub>	q <sub>3</sub>
*q <sub>3</sub>	q <sub>3</sub>	q <sub>3</sub>

**Table 23**

Now it is the transition table of minimized DFA.



**Figure 3.10: Transition diagram of minimized DFA**

---

## 3.6 Summary

---

In this unit you have learnt about the concept of basic Machine, Properties and limitations of Finite State Machine. Moore and mealy Machines, Equivalence of Moore and Mealy machines and Minimization of DFA.

- A system where particular inputs cause particular changes in state can be represented using finite state machines. This example describes the various states of a turnstile. Inserting a coin into a turnstile will unlock it, and after the turnstile has been pushed, it locks again.
- Moore machine is a finite state machine in which the next state is decided by the current state and current input symbol. The output symbol at a given time depends only on the present state of the machine.
- A Mealy machine is a machine in which output symbol depends upon the present input symbol and present state of the machine. In the Mealy machine, the output is represented with each input symbol for each state separated by /.

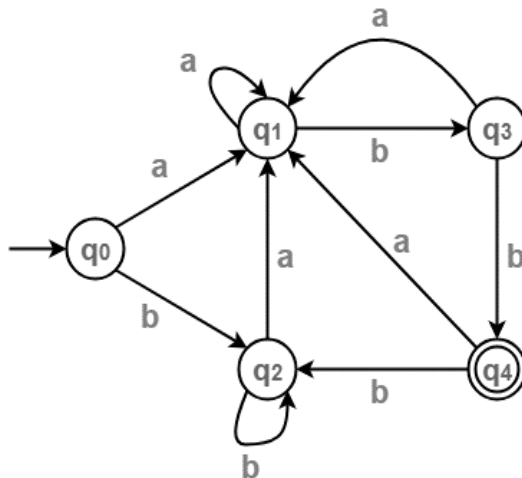
---

## 3.7 Review Questions

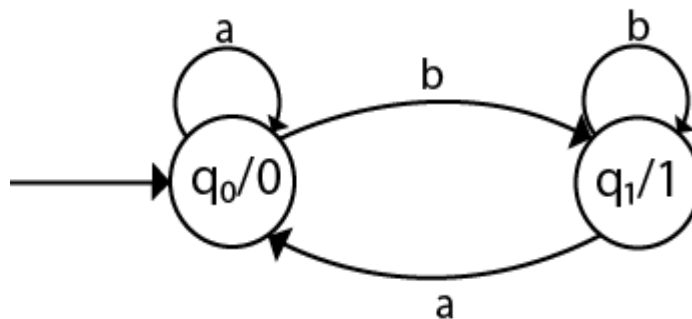
---

Q1.What are finite state machines used for? Is a computer a finite state machine?

- Q2.Design a mealy machine that scans sequence of input of 0 and 1 and generates output 'A' if the input string terminates in 00, output 'B' if the string terminates in 11, and output 'C' otherwise.
- Q3.Design a Moore machine with the input alphabet  $\{0, 1\}$  and output alphabet  $\{Y, N\}$  which produces Y as output if input sequence contains 1010 as a substring otherwise, it produces N as output.
- Q4.Minimize the given DFA-



- Q5.Convert the following Moore machine into its equivalent Mealy machine.



## BIBLIOGRAPHY

<https://medium.com/swlh/introduction-to-deterministic-finite-automata-dfa-40aac64b9895>

<https://www.neuraldump.net/2017/11/nfa-and-dfa-equivalence-theorem-proof-and-example/>







Uttar Pradesh Rajarshi Tandon  
Open University

# **MCS - 113**

## **Master of Computer Science Theory of Computation**

# **Block 2**

## **Regular Expressions and Languages**

---

Unit - 4

<b>Regular Expressions</b>	<b>70</b>
----------------------------	-----------

---

Unit - 5

<b>Properties of Regular Language</b>	<b>89</b>
---------------------------------------	-----------

---

---

## Course Design Committee

---

<b>Prof. Ashutosh Gupta</b> Director (In-charge) School of Computer and Information Sciences, UPRTOU Prayagraj	<b>Chairman</b>
<b>Prof. R. S. Yadav</b> Department of Computer Science and Engineering MNNIT Prayagraj	<b>Member</b>
<b>Dr. Marisha</b> Assistant Professor (Computer Science), School of Sciences, UPRTOU Prayagraj	<b>Member</b>
<b>Mr. Manoj Kumar Balwant</b> Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	<b>Member</b>

---

## Course Preparation Committee

---

<b>Dr. Ravi Shankar Shukla</b> Associate Professor Department of CSE, Invertis University Bareilly-243006, Uttar Pradesh	<b>Author</b>
<b>Prof. Abhay Saxena</b> Professor and Head, Department of Computer Science Dev Sanskriti Vishwavidyalaya, Haridwar, Uttarakhand	<b>Editor</b>
<b>Prof. Ashutosh Gupta</b> Director (In-charge) School of Computer and information, Sciences, UPRTOU Prayagraj	
<b>Mr. Manoj Kumar Balwant</b> Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	<b>Course Coordinator</b>

---

© UPRTOU , Prayagraj - 2023

© MCS - 113 Theory of Computation

ISBN :

---

All Rights are reserved. No Part of this work may reproduced in any form, by mimeograph or any other means, without permission in writing from the Uttar Pradesh Rajarshi Tandon Open University.

Printed and Published by Vinay Kumar Registrar, Uttar Pradesh rajarshi Tandon Open University, Prayagraj - 2023

**Printed By. – M/s K.C.Printing & Allied Works, Panchwati, Mathura -281003.**

---

## Block Introduction

---

This is the second block on Regular Expressions and Languages of theory of automata. The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. A regular expression can also be described as a sequence of pattern that defines a string. Regular expressions are used to match character combinations in strings.

A regular language is a language that can be expressed with a regular expression or a deterministic or non-deterministic finite automata or state machine. A language is a set of String, which are made up of characters from a specified alphabet, or set of symbols.

So, we will begin the first unit on Regular Expressions. In this unit firstly we will discuss about Regular Expressions-Definition, Algebraic Laws of RE, Finite Automata and Regular expressions, Conversion from RE to FA, Conversion from FA to RE, and Arden's Theorem. Therefore, if we talk about the relationship between finite automata and regular expression it says that finite automata are formal (or abstract) machines for recognizing patterns. These machines are used extensively in compilers and text editors, which must recognize patterns in the input. Regular expressions are a formal notation for generating patterns.

Second unit begins with Properties of Regular Language. In this unit you will know all about the pumping lemma for regular sets, applications of the pumping lemma, and closure properties of regular Sets. In the theory of formal languages, the pumping lemma may refer to: Pumping lemma for regular languages, the fact that all sufficiently long strings in such a language have a substring that can be repeated arbitrarily many times, usually used to prove that certain languages are not regular.

As you study the material, you will find that figures, tables are properly used and these will help to understand the concept. There are many sections in the units to easily understand the topic. Every unit has summary and review questions in the end of the unit, which will help you to review yourself.

In your study, you will find that every unit has different equal length and your study time will vary for each unit.

We hope you will enjoy studying the material and once again wish you all the best for your success.

---

# UNIT-4 Regular Expressions

---

## Structure

- 4.0 Introduction
- 4.1 Regular Expressions-Definition
- 4.2 Algebraic Laws of RE
- 4.3 Finite Automata and Regular expressions
- 4.4 Conversion from RE to FA
- 4.5 Conversion from FA to RE
- 4.6 Arden's Theorem
- 4.7 Summary
- 4.8 Review Questions

## 4.0 Introduction

This is the first unit of this block. In this block there are two units. In this unit, you will learn about regular expression. In the section 1.1 you will know about definition of regular expressions, Algebraic Laws of RE defines in the section 1.2, Next section i.e. section 1.3 defines Finite Automata and Regular expressions. There is another very important section i.e. section 1.4 explain conversion from RE to FA. You will know the details about conversion from FA to RE. Arden's Theorem is defined in the section 1.6. Summary and Review question has shown in the section 1.7 and 1.8 respectively.

### Objective

After studying this unit, you should be able to explain:

- Definition of Regular Expressions
- Algebraic Laws of RE
- Finite Automata and Regular expressions

- Conversion from RE to FA
- Conversion from FA to RE
- Arden's Theorem.

## 4.1 Regular Expressions Definition

- The language recognized by finite automata can also be described by simple expressions known as Regular Expressions. It is the most effective and efficient way to represent any language.
- The languages recognized by some regular expression are stated as Regular languages.
- A regular expression can also be termed as a sequence of pattern that defines a string.
- Another important use of Regular expressions is to match character combinations in strings. This pattern can be used by String searching algorithm to find the operations on a string.

### For instance:

In a regular expression,  $x^*$  means zero or more occurrence of  $x$ . It can generate  $\{e, x, xx, xxx, xxxx, \dots\}$

In a regular expression,  $x^+$  means one or more occurrence of  $x$ . It can generate  $\{x, xx, xxx, xxxx, \dots\}$

### Formal Definition

An expression  $R$  is a regular expression if  $R$  is

1.  $a$  for some  $a$  in some alphabet  $\Sigma$
2.  $\epsilon$ ,
3.  $\emptyset$ ,
4.  $(R_1 \cup R_2)$  for some regular expressions  $R_1$  and  $R_2$ ,
5.  $(R_1 \circ R_2)$  for some regular expressions  $R_1$  and  $R_2$ , or
6.  $(R_1)^*$  for some regular expression  $R_1$ .

When the meaning is clear from the context,  $()$  and can be removed from the expression.

### Operations on Regular Language

The various operations on regular language are:

**Union:** If  $L$  and  $M$  are two regular languages then their union  $L \cup M$  is also a union.

$$1. L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$$

**Intersection:** If L and M are two regular languages then their intersection is also an intersection.

$$1. L \cap M = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$$

**Kleen closure:** If L is a regular language then its Kleen closure  $L^*$  will also be a regular language.

$$1. L^* = \text{Zero or more occurrence of language } L.$$

**Example:**

*Write the regular expression for the language accepting all combinations of a's, over the set  $\Sigma = \{a\}$*

**Solution:**

All combinations of a's mean a may be zero, single, double and so on. If a is appearing zero times, that means a null string. That is, we expect the set of  $\{\epsilon, a, aa, aaa, \dots\}$ . So we give a regular expression for this as:

$$R = a^*$$

That is Kleen closure of a.

## 4.2 Algebraic Laws of RE

**Associative Law:**

It is the property of an operator that allows us to regroup the operands when operator is applied twice.

An example for arithmetic associative law is

$$(X+Y) + Z = X+(Y+Z)$$

Similarly, for an RE, the associative law for union is  $(L+M) + N = L+(M+N)$   
The associative law for concatenation states that  $(LM)N = L(MN)$

**Commutative Law:**

It is the property of an operator that allows us to switch the order of its operand and get the same result.

An example for arithmetic commutative law is

$$X+Y=Y+X$$

Similarly, for an RE, the commutative law for union is  $L+M=M+L$

But, the commutative law for concatenation does not hold good, i.e.,  $LM \neq ML$

### **Distributive Law:**

It involves two operators and asserts that one operator can be pushed down to be applied to each argument of other operator individually.

An example for arithmetic distributive law is

$$X*(Y+Z) = X*Y+X*Z$$

In the case of RE, the distributive law is stated in two forms, that are, Left Distributive law of concatenation over union, which states that

$$L(M+N) = LM+LN$$

### **Idempotent law:**

An operator is said to be idempotent if the result of applying it to two of the same values as arguments is that value itself. This law is not analogous to arithmetic laws.

As an example,  $X+X \neq X$  *or*,  $X*X \neq X$

Although there are some cases for the values of X when the equality holds true such as  $0+0 = 0$

For an RE, the idempotent law for union is  $L+L=L$  which means if we take two identical copies of an expression, we can replace them by only one copy of the expression.

### **Identity Law:**

Let us see the Identity for Regular Expressions.

$$\emptyset + A = A + \emptyset = A \text{ and } \epsilon.A = A.\epsilon = A$$

### **Annihilator Law:**

Let us see the Annihilator for Regular Expressions

$$\emptyset.A = A.\emptyset = \emptyset$$

### **Closure Laws:**

Let us see the Closure Laws for Regular Expressions.

$(A^*)^* = A^*$ ,  $\emptyset^* = \epsilon$ ,  $\epsilon^* = \epsilon$ ,  $A^+ = AA^* = A^*A$ , and  $A^* = A^+ \cup \epsilon$ .

**De Morgan Law:**

$$(L + B)^* = (L^*B^*)^*$$

### 4.3 Finite Automata and Regular expressions

Given any regular expression as a pattern for string searching, we might want to change this pattern into a deterministic finite automaton (DFA) or nondeterministic finite automaton (NDFA) for effective string searching; a deterministic automaton only has to scan each input symbol once.

Theorem 1.1 If  $L_1 = L(M_1)$  and  $L_2 = L(M_2)$  for languages  $L_i \subseteq \Sigma^*$  then

1. There is a mechanism for machine  $M$  recognizing  $L_1 \cup L_2$
2. There is an automaton for machine  $M$  recognizing  $L_1 \circ L_2$
3. There is an automaton recognizing the language  $L_1^*$
4. There is an automaton recognizing  $\Sigma^* - L_1$
5. There is an automaton recognizing  $L_1 \cap L_2$
6. If  $a \in \Sigma$  then there is an automaton recognizing  $\{a\}$
7. There is an automaton recognizing  $\emptyset$ .

From all of the above points it follows that if  $A$  is a regular language then there is a finite automaton recognizing  $A$ .

For example, justify why there would be a finite automaton recognizing the language represented by  $a \cup (ab)^*$ .

Proof: We will proof for nondeterministic automata as we know that deterministic and nondeterministic automata are of equivalent power.

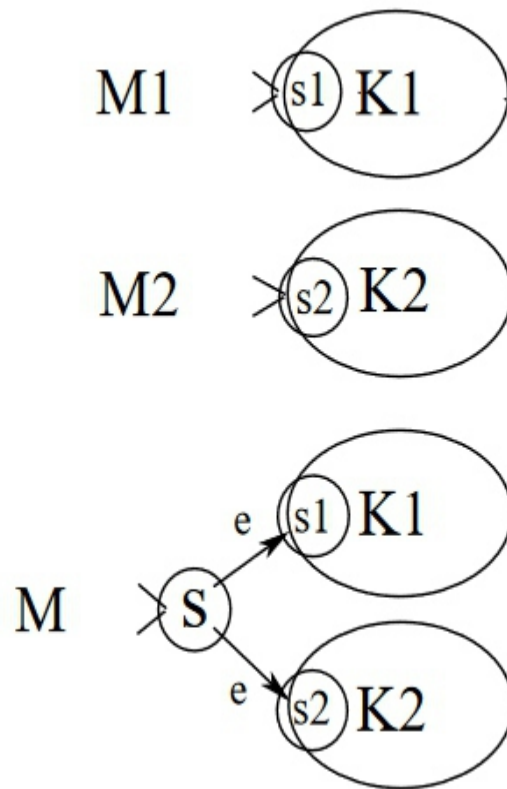
#### 4.3.1 Union

For union, suppose  $M_1$  is  $(K_1, \Sigma, \Delta_1, s_1, F_1)$  and  $M_2$  is  $(K_2, \Sigma, \Delta_2, s_2, F_2)$ . Then let  $M$  be  $(K, \Sigma, \Delta, s, F)$  where

$$\begin{aligned} K &= K_1 \cup K_2 \cup \{s\} \\ F &= F_1 \cup F_2 \\ \Delta &= \Delta_1 \cup \Delta_2 \cup \{(s, e, s_1), (s, e, s_2)\} \end{aligned}$$

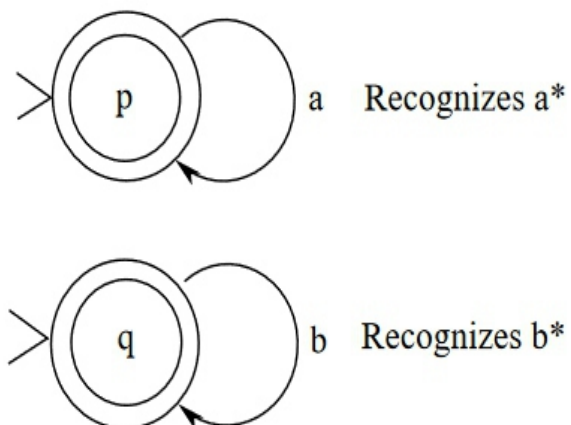


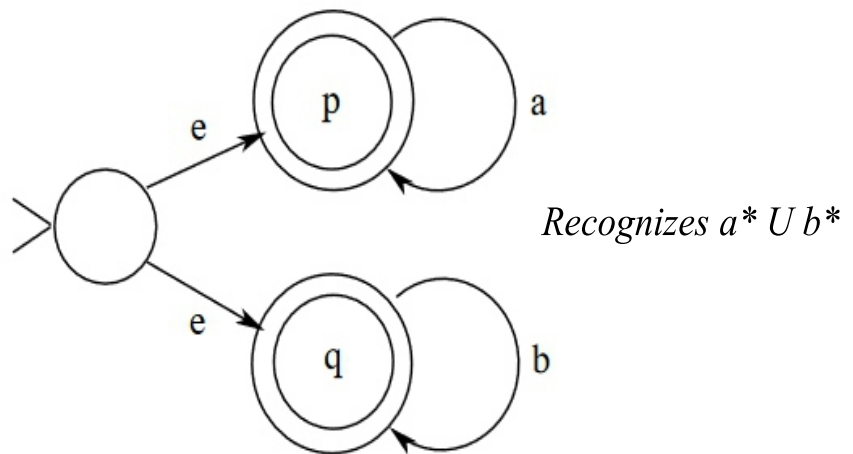
And  $s$  is a new state. Then  $L(M) = L(M_1) \cup L(M_2)$ . Diagram:



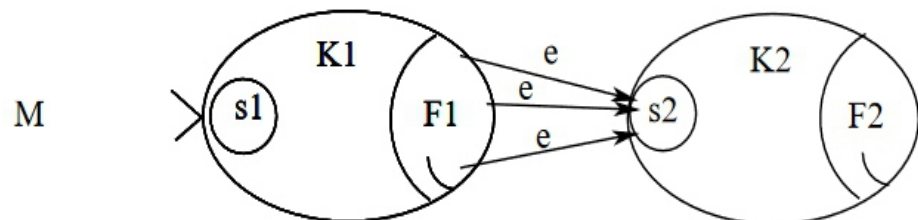
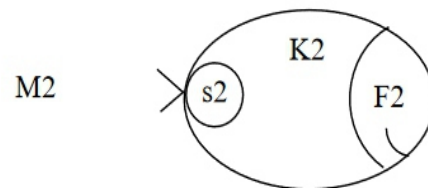
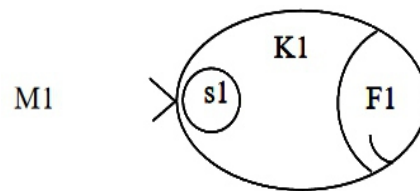
One important point must be noted that  $\epsilon$  arrows are suitable for this construction.

**Example:**



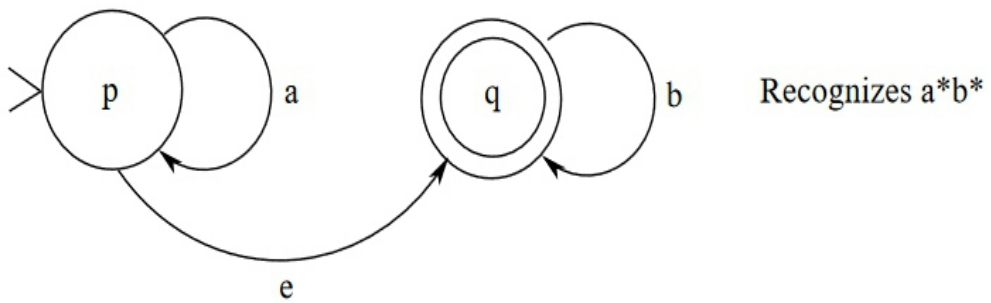
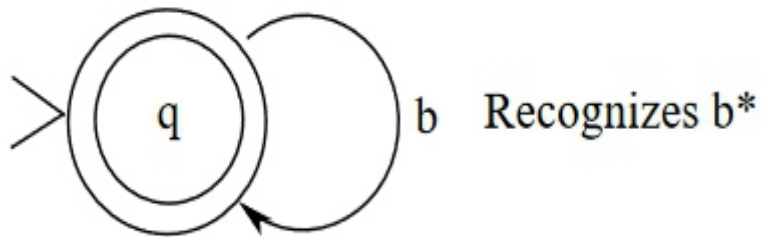
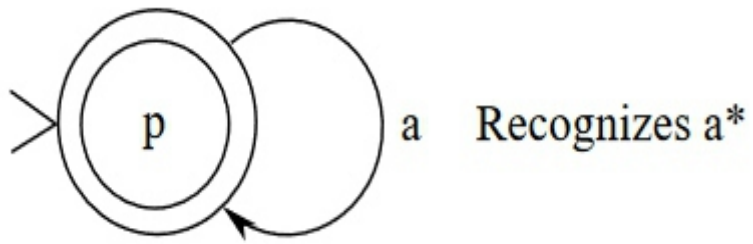


#### 4.3.2 Concatenation

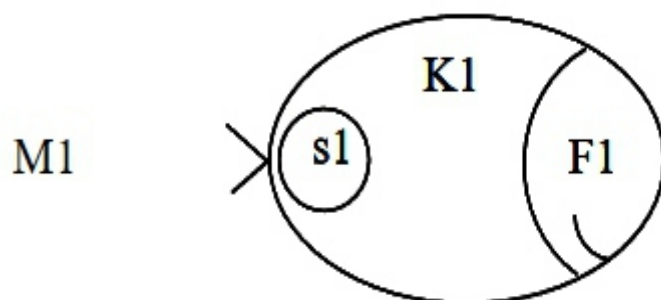


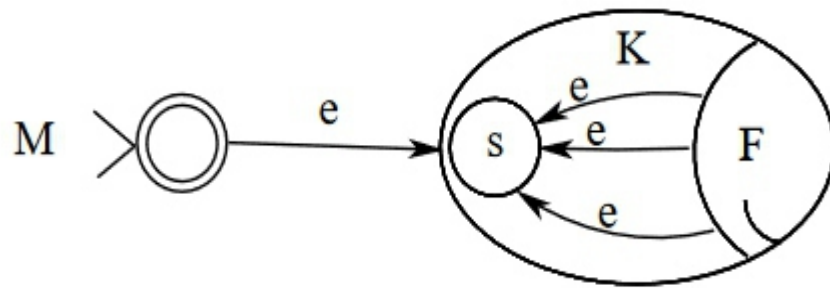
The states in  $F_1$  are no longer accepting states. Then  $L(M) = L(M_1) \circ L(M_2)$

**Example:**



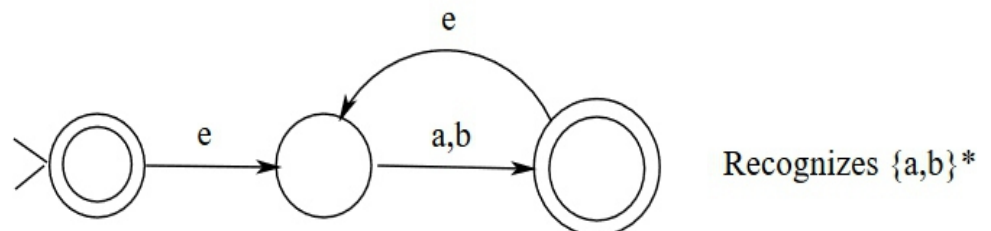
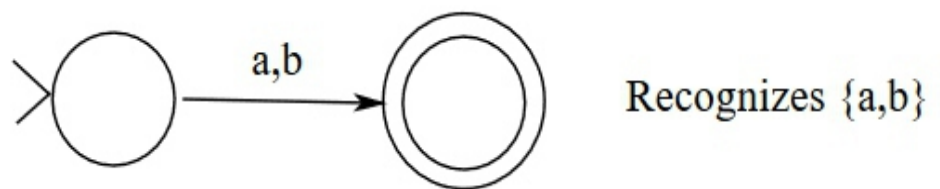
#### 4.3.3 Kleene star





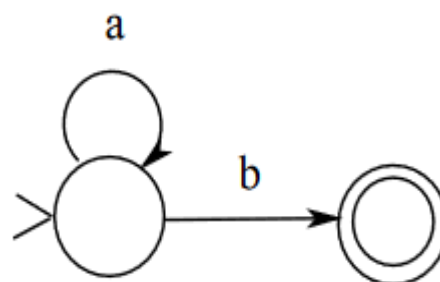
Then  $L(M) = L(MI)^*$

**Example:**



Now the question arise here how would you modify this automaton to recognize  $\{a, b\}^+$ ?

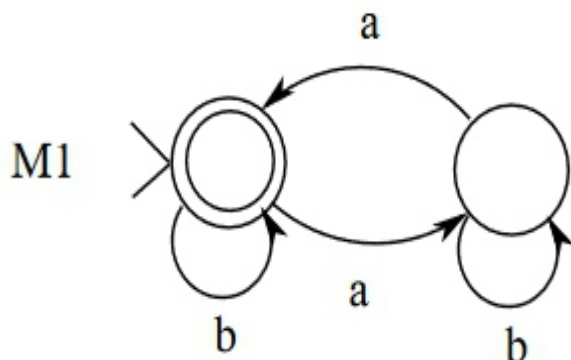
The solution is shown blow:



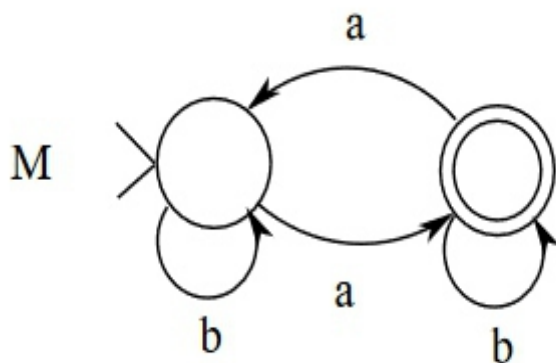
### 4.3.4 Complementation

Let  $M_1 = (K, \Sigma, \delta, s, F)$  be a deterministic finite automaton. Let  $M$  be  $(K, \Sigma, \delta, s, K - F)$ . Then  $L(M) = \Sigma^* - L(M_1)$ .

**Example:**

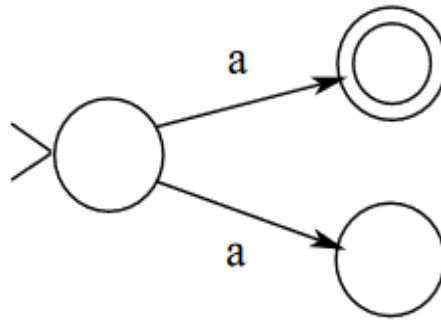


Recognizes strings with even number of a's



Recognizes strings with odd number of a's

Why does the automaton have to be deterministic for this to work? An example showing that  $M_1$  has to be deterministic for this construction to work:



#### 4.3.5 Intersection

If there are two types of context free languages  $L_1$  and  $L_2$ , their intersection  $L_1 \cap L_2$  need not be context free. For example,

$L_1 = \{a^n b^n c^m \mid n \geq 0 \text{ and } m \geq 0\}$  and  $L_2 = \{a^m b^n c^n \mid n \geq 0 \text{ and } m \geq 0\}$

$L_3 = L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$  need not be context free.

$L_1$  says number of a's and the number of b's should be equal  $L_2$  says number of b's and number of c's should be equal. Their intersection says both the conditions must be true, but push down automata can compare only two. So it cannot be accepted by pushdown automata, hence not context free.

---

### Check your progress

---

Q1. Define all the algebraic laws of Regular expression with suitable example.

Q2. What is the relation between finite automata and regular expression? Elaborate your answer.

## 4.4 Conversion from RE to FA

In this topic we will learn about the conversion from regular expression to finite automata. For this purpose, we will use a method called subset method. With the help of this method, we can acquire finite automata (FA) from the given regular expression. The method is given below:

Step 1: Design a transition diagram for given regular expression, using NFA with  $\epsilon$  moves.

Step 2: Convert this NFA with  $\epsilon$  to NFA without  $\epsilon$ .

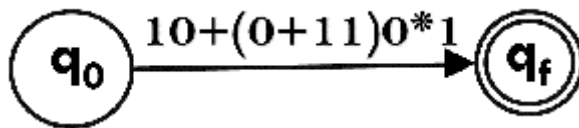
Step 3: Convert the obtained NFA to equivalent DFA.

**Example:**

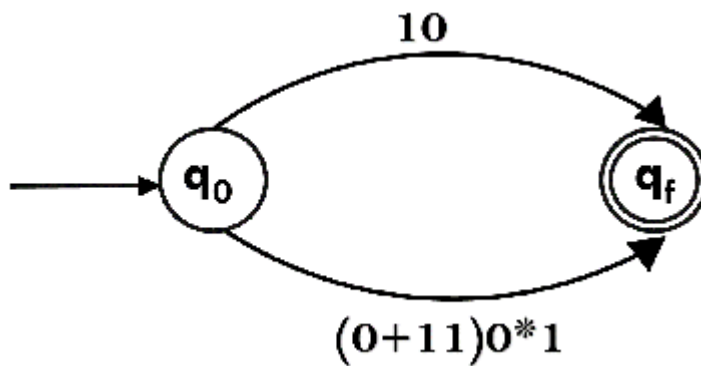
Design a FA from given regular expression  $10 + (0 + 11)0^*1$ .

**Solution:** First we will construct the transition diagram for a given regular expression.

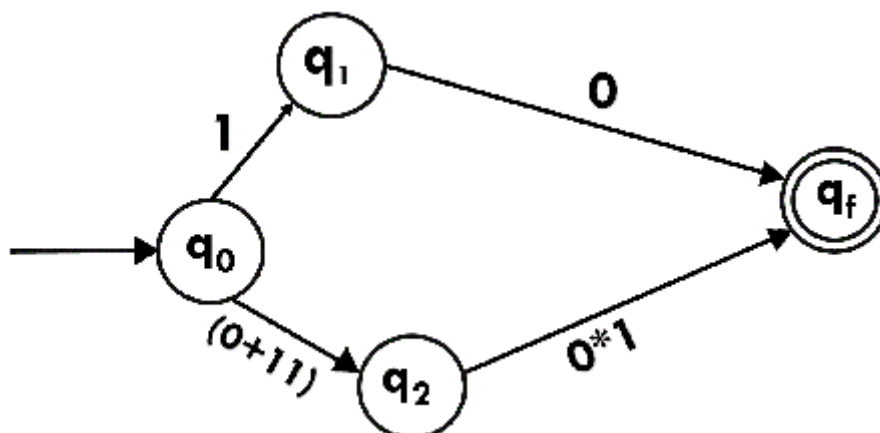
**Step 1:**



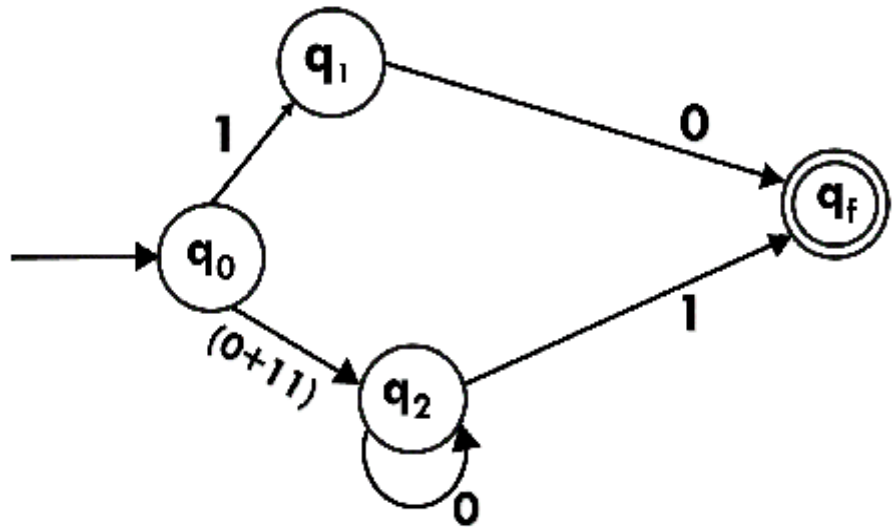
**Step 2:**



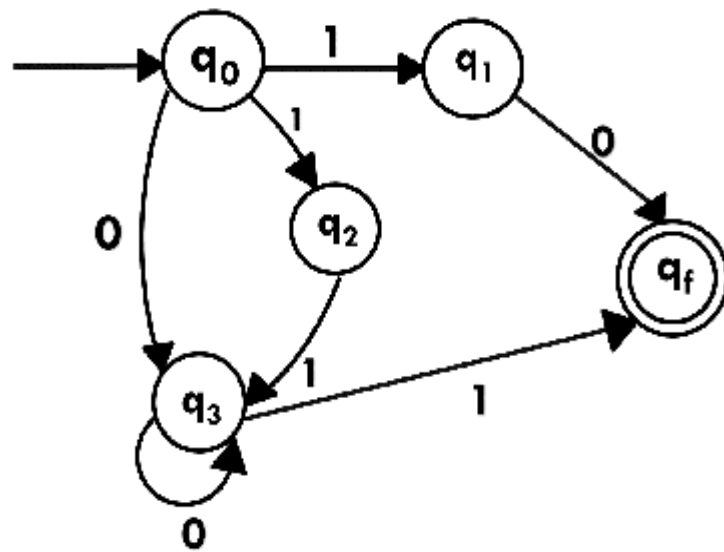
**Step 3:**



Step 4:



Step 5:



Now we have got NFA without  $\epsilon$ . The next step will be the conversion from NFA to required DFA for that, we will first write a transition table for this NFA.

State	0	1
$\rightarrow q_0$	$q_3$	$\{q_1, q_2\}$
$q_1$	$q_f$	$\Phi$
$q_2$	$\phi$	$q_3$
$q_3$	$q_3$	$Q_f$
$*q_f$	$\phi$	$\Phi$



The equivalent DFA will be:

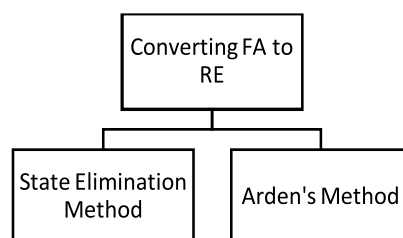
State	0	1
$\rightarrow[q0]$	$[q3]$	$[q1, q2]$
$[q1]$	$[qf]$	$\phi$
$[q2]$	$\phi$	$[q3]$
$[q3]$	$[q3]$	$[qf]$
$[q1, q2]$	$[qf]$	$[qf]$
$*[qf]$	$\phi$	$\phi$

In this unit you have learnt about Context Free Grammar and its properties. You have also learnt about sentential forms, derivations of left most and right most, and the language of context free grammar.

- **Context Free Grammars or CFGs** define a formal language. Formal languages work strictly under the defined rules and their sentences are not influenced by the context.
  - Every string of symbols in the derivation is a sentential form
  - A sentence is a sentential form that has only terminal symbols
  - A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded next in the derivation
  - A rightmost derivation works right to left instead
  - Some derivations are neither leftmost nor rightmost

## 4.5 Conversion from FA to RE

The two popular methods for converting a DFA to its regular expression are-



1. State Elimination Method
2. Arden's Method

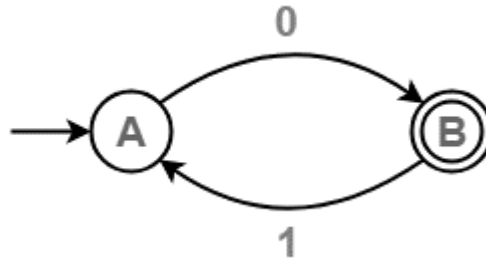
### State Elimination Method-

Step1: If the start state is an accepting state or has transitions in, add a new non-accepting start state and add an  $\epsilon$ -transition between the new start state and the former start state.

Step2: If there is more than one accepting state or if the single accepting state has transitions out, add a new accepting state, make all other states non-accepting, and add an  $\epsilon$ -transition from each former accepting state to the new accepting state.

Step3: For each non-start non-accepting state in turn, eliminate the state and update transitions accordingly.

**Example: Find regular expression for the following DFA-**

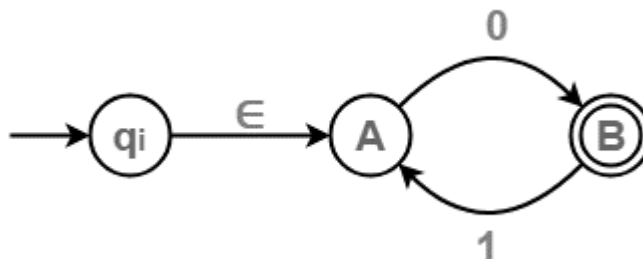


**Solution-**

Step-01:

- Initial state A has an incoming edge.
- So, we create a new initial state  $q_i$ .

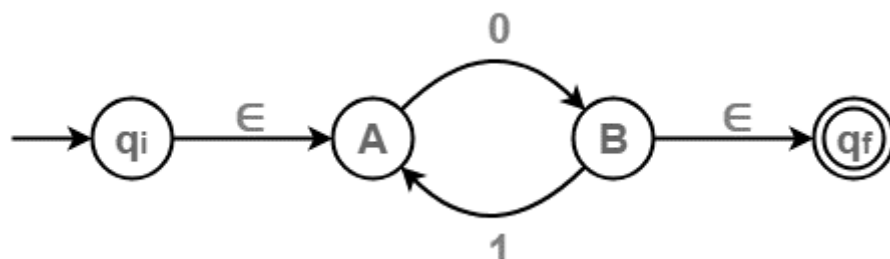
The resulting DFA is-



Step-02:

- Final state B has an outgoing edge.
- So, we create a new final state  $q_f$ .

The resulting DFA is-



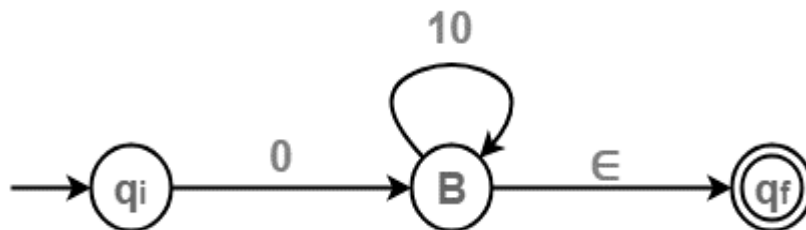
Step-03:

Now, we start eliminating the intermediate states.

First, let us eliminate state A.

- There is a path going from state  $q_i$  to state B via state A.
- So, after eliminating state A, we put a direct path from state  $q_i$  to state B having cost  $\epsilon.0 = 0$
- There is a loop on state B using state A.
- So, after eliminating state A, we put a direct loop on state B having cost  $1.0 = 10$ .

Eliminating state, A, we get-

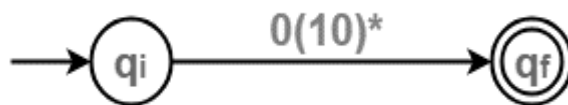


Step-04:

Now, let us eliminate state B.

- There is a path going from state  $q_i$  to state  $q_f$  via state B.
- So, after eliminating state B, we put a direct path from state  $q_i$  to state  $q_f$  having cost  $0(10)^* \epsilon = 0(10)^*$

Eliminating state B, we get-



From here,

Regular Expression =  $0(10)^*$

## 4.6 Arden's Theorem

The Arden's Theorem is useful for checking the equivalence of two regular expressions as well as in the conversion of DFA to a regular expression.

Let us see its use in the conversion of DFA to a regular expression.

Following algorithm is used to build the regular expression form given DFA.

1. Let  $q_1$  be the initial state.
2. There are  $q_2, q_3, q_4 \dots q_n$  number of states. The final state may be some  $q_j$  where  $j \leq n$ .
3. Let  $\alpha_{ji}$  represents the transition from  $q_j$  to  $q_i$ .
4. Calculate  $q_i$  such that

$$q_i = \alpha_{ji} * q_j$$

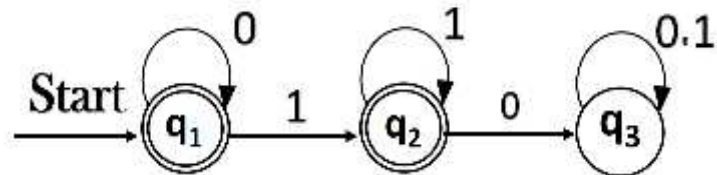
If  $q_j$  is a start state, then we have:

$$q_i = \alpha_{ji} * q_j + \epsilon$$

5. Similarly, compute the final state which ultimately gives the regular expression 'r'

**Example:**

**Construct the regular expression for the given DFA**



**Solution:**

Let us write down the equations

$$q_1 = q_1 0 + \epsilon$$

Since  $q_1$  is the start state, so  $\epsilon$  will be added, and the input 0 is coming to  $q_1$  from  $q_1$  hence we write

State = source state of input  $\times$  input coming to it

Similarly,

$$q_2 = q_1 1 + q_2 1$$

$$q_3 = q_2 0 + q_3 (0+1)$$

Since the final states are  $q_1$  and  $q_2$ , we are interested in solving  $q_1$  and  $q_2$  only.

Let us see  $q_1$  first

$$q_1 = q_1 0 + \varepsilon$$

We can re-write it as

$$q_1 = \varepsilon + q_1 0$$

Which is similar to  $R = Q + RP$ , and gets reduced to  $R = QP^*$ .

Assuming  $R = q_1$ ,  $Q = \varepsilon$ ,  $P = 0$

We get

$$q_1 = \varepsilon. (0)^*$$

$$q_1 = 0^* (\varepsilon.R^* = R^*)$$

Substituting the value into  $q_2$ , we will get

$$q_2 = 0^* 1 + q_2 1$$

$$q_2 = 0^* 1 (1)^* (R = Q + RP \rightarrow QP^*)$$

**The regular expression is given by**

$$r = q_1 + q_2$$

$$= 0^* + 0^* 1.1^*$$

$$r = 0^* + 0^* 1 + (1.1^* = 1+)$$

---

## 4.7 Summary

---

In this unit you have learnt about Regular Expressions definition, Algebraic Laws of Regular Expression, Finite Automata and Regular expressions, Conversion from Regular Expression to Finite Automata, Conversion from Finite Automata to Regular Expression and Arden's Theorem.

- A simple example for a regular expression is a (literal) string. For example, the Hello World regex matches the "Hello World" string.
- . (Dot) is another example for a regular expression. A dot matches any single character; it would match, for example, "a" or "1".
- Short for regular expression, a regex is a string of text that allows you to create patterns that help match, locate, and manage text. Perl is a great example of a programming language that utilizes regular expressions.

- A finite automaton (FA) is a simple idealized machine used to recognize patterns within input taken from some character set (or alphabet)  $C$ .
- The job of an FA is to accept or reject an input depending on whether the pattern defined by the FA occurs in the input.

---

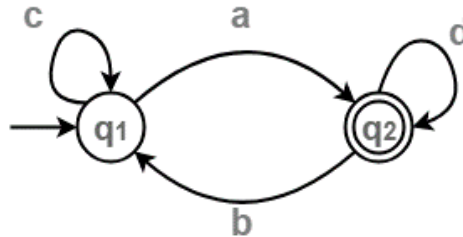
## 4.8 Review Questions

---

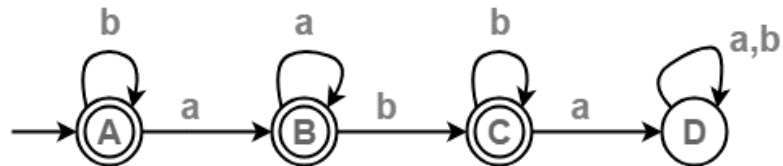
Q1. Design a NFA from given regular expression  $1(1^*01^*01^*)^*$ .

Q2. Construct the FA for regular expression  $0^*1 + 10$ .

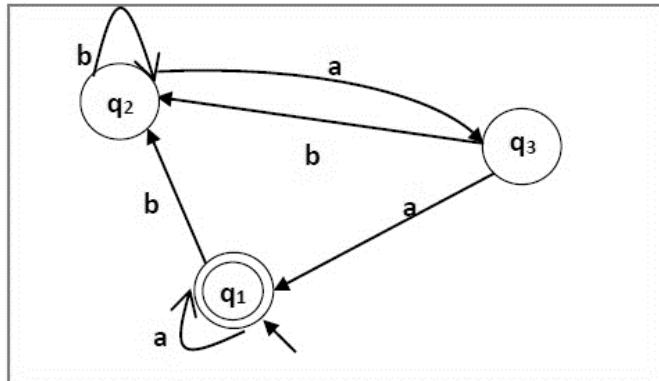
Q3. Find regular expression for the following DFA-



Q4. Find regular expression for the following DFA-



Q5. Construct a regular expression corresponding to the automata given below with the help of Arden's Theorem-



Q6. Construct a DFA with reduced states equivalent to the R.E.

$10 + (0+11)^*0^*1$ .

---

# UNIT-5 Properties of Regular Language

---

## Structure

- 5.0 Introduction
- 5.1 The Pumping Lemma for Regular Sets
- 5.2 Applications of the pumping lemma
- 5.3 Closure properties of regular sets.
- 5.4 Summary
- 5.5 Review Questions

## 5.0 Introduction

This is the second unit of this block. This unit is divided in to five sections and explain all the properties of regular language. In the section 2.1, you will learn about the Pumping Lemma for Regular Sets. Section 2.2 provide Applications of the pumping lemma. Closure properties of regular sets define in the Section 2.3. Section 2.4 has Summary and Review Questions is providing in the Section 2.5.

### Objective

After studying this unit, you should be able to define:

- The Pumping Lemma for Regular Sets
- Applications of the pumping lemma
- Closure properties of regular sets.

## 5.1 The Pumping Lemma for Regular Sets

In the theory of formal languages, the pumping lemma for regular languages is a lemma that describes an essential property of all regular languages. Informally, it says that all sufficiently long words in a regular language may be pumped—that is, have a middle section of the word repeated an arbitrary number of times—to produce a new word that also lies within the same language.

Specifically, the pumping lemma says that for any regular language  $L$  there exists a constant  $p$  such that any word  $w$  in  $L$  with length at least  $p$  can be split

into three substrings,  $w = x y z$ , where the middle portion  $y$  must not be empty, such that the words  $x z, x y z, x y y z, x y y y z, \dots$  constructed by repeating  $y$  zero or more times are still in  $L$ . This process of repetition is known as "pumping". Moreover, the pumping lemma guarantees that the length of  $xy$  will be at most  $p$ , imposing a limit on the ways in which  $w$  may be split. Finite languages vacuously satisfy the pumping lemma by having  $p$  equal to the maximum string length in  $L$  plus one.

Let  $L$  be a regular language. Then there exists an integer  $p \geq 1$  depending only on  $L$  such that every string  $w$  in  $L$  of length at least  $p$  ( $p$  is called the "pumping length") can be written as  $w = x y z$  (i.e.,  $w$  can be divided into three substrings), satisfying the following conditions:

- $|y| \geq 1$
- $|xy| \leq p$
- $(\forall n \geq 0) (xy^n z \in L)$

$y$  is the substring that can be pumped (removed or repeated any number of times, and the resulting string is always in  $L$ ).

(1) Means the loop  $y$  to be pumped must be of length at least one;

(2) Means the loop must occur within the first  $p$  characters.  $|x|$  must be smaller than  $p$  (conclusion of (1) and (2)), but apart from that, there is no restriction on  $x$  and  $z$ . In simple words, for any regular language  $L$ , any sufficiently long word  $w$  (in  $L$ ) can be split into 3 parts. i.e.  $w=xyz$ , such that all the strings  $xy^n z$  for  $n \geq 0$  are also in  $L$ . Below is a formal expression of the Pumping Lemma.

$$\forall L \subseteq \Sigma^*$$

$$(regular(L) \Rightarrow$$

$$((\exists p \geq 1) ((\forall w \in L) ((|w| \geq p) \Rightarrow$$

$$((\exists x, y, z \in \Sigma^*) (w = xyz \wedge (|y| \geq 1 \wedge |xy| \leq p \wedge (\forall n \geq 0) (xy^n z \in L))))))$$

### The Pumping Lemma: Examples

Consider the following three languages:

$$L_1 = \{a^n b^n \mid 0 \leq n \leq 100\}$$

$$L_2 = \{a^n b^n \mid n \geq 0\}$$

$$L_3 = \{a^n b^m \mid n, m \geq 0\}$$

The first language is regular, since it contains only a finite number of strings.

The third language is also regular, since it is equivalent to the regular expression  $(a^*)(b^*)$ .

The second language consists of all strings which contain a number of 'a's followed by an equal number of 'b's.



### Lemma: $L_2$ is not regular

Strategy: Proof by contradiction

Proof: Let us assume that it is regular; then we must have some set of strings of the form

$$\{xy^iz \mid i \geq 0\} \subseteq L$$

Suppose such a subset did exist.

- Obviously,  $y$  could not contain a mixture of a's and b's, since this would mean that  $xy^iz$  would have 'b's before 'a's. Thus,  $y$  must consist solely of a's or solely of 'b's.
- Let us assume then that  $y$  consists solely of 'a's. Then if, for some  $n$  we have that  $xy^n z$  is in the language, there is no way that  $xy^{n+1} z$  can be in the language, since this will contain extra 'a's without any extra 'b's. Thus there will be no way for  $xy^i z$  to be in the language for every  $i \geq 0$ .
- We can use exactly the same argument to refute the supposition that  $y$  can consist entirely of 'b's.

Thus we have shown that  $y$  cannot consist of 'a's 'b's or their mixture; i.e. no such  $y$  exists, and so the Pumping Lemma is not satisfied. Thus  $L_2$  is not regular.

## 5.2 Applications of the pumping lemma

Not all languages are regular. For example, the language  $L = \{a^n b^n : n \geq 0\}$  is not regular. Similarly, the language  $\{a^p : p \text{ is a prime number}\}$  is not regular. A pertinent question therefore is how do we know if a language is not regular.

Proving languages non-regular

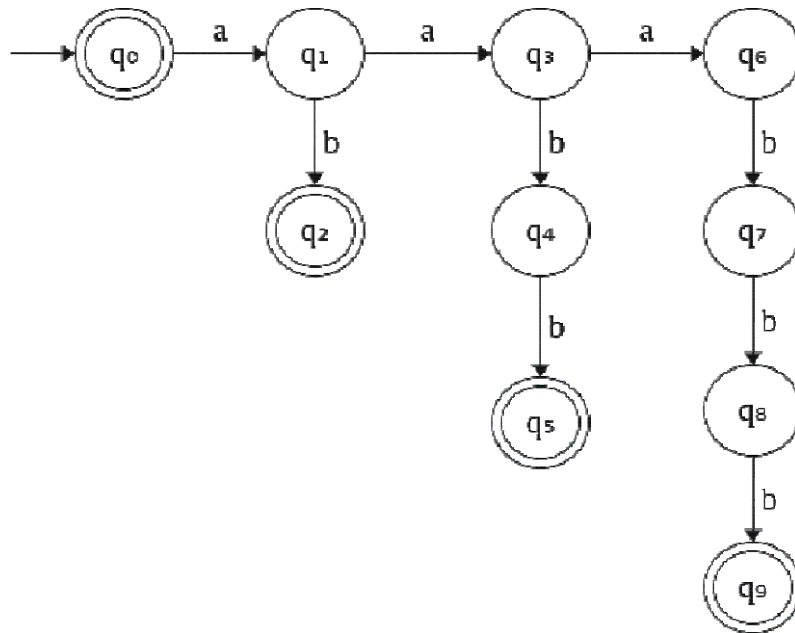
### 1. *The language $L = \{a^n b^n : n \neq 0\}$ is not regular.*

Before proving  $L$  is not regular using pumping property, let's see why we can't come up with a DFA or regular expression for  $L$ .

$$L = \{\epsilon, ab, aabb, aaabbb, \dots\}$$

It may be tempting to use the regular expression  $a^*b^*$  to describe  $L$ . No doubt,  $a^*b^*$  generates these strings. However, it is not appropriate since it generates other strings not in  $L$  such as  $a$ ,  $b$ ,  $aa$ ,  $ab$ ,  $aaa$ ,  $aab$ ,  $abb$ , ...

Let's try to come up with a DFA. Since it has to accept  $\epsilon$ , start state has to be final. The following DFA can accept  $a^n b^n$  for  $n \leq 3$ . i.e.  $\{\epsilon, a, b, ab, aabb, aaabbb\}$



The basic problem is DFA does not have any memory. A transition just depends on the current state. So it cannot keep count of how many a's it has seen. So, it has no way to match the number of a's and b's. So, only way to accept all the strings of L is to keep adding newer and newer states which makes automaton to infinite states since n is unbounded.

Now, let's prove that L does not have the pumping property.

Let's assume L is regular. Let p be the pumping length.

Consider a string  $w = aa....abb....b$  such that  $|w| = p$ .

$$\Rightarrow w = a^{p/2}b^{p/2}$$

We know that w can be broken into three terms xyz such that  $y \neq \epsilon$  and  $xy^iz \in L$ .

There are three cases to consider.

- Case 1: y is made up of only a's Then  $xy^2z$  has more a's than b's and does not belong to L.
- Case 2: y is made up of only b's Then  $xy^2z$  has more b's than a's and does not belong to L.
- Case 3: y is made up of a's and b's Then  $xy^2z$  has a's and b's out of order and does not belong to L.

Since none of the 3 cases hold, the pumping property does not hold for L. And **therefore L is not regular**.

Let us assume  $L$  is regular. Let  $p$  be the pumping length.

Consider a string  $w = a^p b b a^p$ .

$$|w| = 2p + 2 \geq p$$

Since,  $|xy| \leq p$ ,  $xy$  will consist of only a's.

$\Rightarrow y$  is made of only a's

$\Rightarrow y^2$  is made of more number of a's than  $y$  since  $|y| > 0$

(Let's say  $y^2$  has  $m$  a's more than  $y$  where  $m > 1$ )

$\Rightarrow xy^2z = a^{p+m} b b a^p$  where  $m \geq 1$

$\Rightarrow xy^2z = a^{p+m} b b a^p$  cannot belong to  $L$ .

Therefore, pumping property does not hold for  $L$ . Hence,  **$L$  is not regular.**

**3. The language  $L = \{a^n : n \text{ is prime}\}$  is not regular.**

Let's assume  $L$  is regular. Let  $p$  be the pumping length. Let  $q \geq p$  be a prime number (since we cannot assume that pumping length  $p$  will be prime).

Consider the string  $w = a a \dots a$  such that  $|w| = q \geq p$ .

We know that  $w$  can be broken into three terms  $xyz$  such that  $y \neq \epsilon$  and  $xyz \in L$

$\Rightarrow xy^{q+1}z$  must belong to  $L$

$\Rightarrow |xy^{q+1}z|$  must be prime

$$\begin{aligned} |xy^{q+1}z| &= |xyz y^q| \\ &= |xyz| + |y^q| \\ &= q + q \cdot |y| \\ &= q(1 + |y|) \text{ which is a composite number.} \end{aligned}$$

Therefore,  $xy^{q+1}z$  cannot belong to  $L$ . Hence,  **$L$  is not regular.**

---

## Check your progress

---

Q1. Define Pumping Lemma for Regular Sets using suitable example.

Q2. What are the Applications of the pumping lemma? Elaborate your answer with suitable example.

## 5.3 Closure properties of regular sets.

Closure properties on regular languages are defined as certain operations on regular language which are guaranteed to produce regular language. Closure refers to some operation on a language, resulting in a new language that is of same “type” as originally operated on i.e., regular.

Regular languages are closed under following operations.

Consider  $L$  and  $M$  are regular languages:

**1. Kleen Closure:**

RS is a regular expression whose language is  $L$ ;  $M$ .  $R^*$  is a regular expression whose language is  $L^*$ .

**2. Positive closure:**

RS is a regular expression whose language is  $L$ ,  $M$ .  $R^+$  is a regular expression whose language is  $L^+$ .

**3. Complement:**

The complement of a language  $L$  (with respect to an alphabet  $E$  such that  $E^*$  contains  $L$ ) is  $E^* - L$ . Since  $E^*$  is surely regular, the complement of a regular language is always regular.

**4. Reverse Operator:**

Given language  $L$ ,  $L^R$  is the set of strings whose reversal is in  $L$ .

Example:  $L = \{0, 01, 100\}$ ;

$L^R = \{0, 10, 001\}$ .

**Proof:** Let  $E$  be a regular expression for  $L$ . We show how to reverse  $E$ , to provide a regular expression  $E^R$  for  $L^R$ .

**5. Union:**

Let  $L$  and  $M$  be the languages of regular expressions  $R$  and  $S$ , respectively. Then  $R+S$  is a regular expression whose language is  $(L \cup M)$ .

**6. Intersection:**

Let  $L$  and  $M$  be the languages of regular expressions  $R$  and  $S$ , respectively then it a regular expression whose language is  $L \cap M$ .

**Proof:** Let  $A$  and  $B$  be DFA's whose languages are  $L$  and  $M$ , respectively. Construct  $C$ , the product automaton of  $A$  and  $B$  make the final states of  $C$  be the pairs consisting of final states of both  $A$  and  $B$ .

**7. Set Difference operator:**

If  $L$  and  $M$  are regular languages, then so is  $L - M =$  strings in  $L$  but not  $M$ .

**Proof:** Let  $A$  and  $B$  be DFA's whose languages are  $L$  and  $M$ , respectively. Construct  $C$ , the product automaton of  $A$  and  $B$  make the final states of  $C$  be the pairs, where  $A$ -state is final but  $B$ -state is not.

## 8. Homomorphism:

A homomorphism on an alphabet is a function that gives a string for each symbol in that alphabet. Example:  $h(0) = ab$ ;  $h(1) = E$ . Extend to strings by  $h(a_1 \dots a_n) = h(a_1) \dots h(a_n)$ . Example:  $h(01010) = ababab$ .

If  $L$  is a regular language, and  $h$  is a homomorphism on its alphabet, then  $h(L) = \{h(w) \mid w \text{ is in } L\}$  is also a regular language.

**Proof:** Let  $E$  be a regular expression for  $L$ . Apply  $h$  to each symbol in  $E$ . Language of resulting  $R$ ,  $E$  is  $h(L)$ .

## 9. Inverse Homomorphism:

Let  $h$  be a homomorphism and  $L$  a language whose alphabet is the output language of  $h$ .  $h^{-1}(L) = \{w \mid h(w) \text{ is in } L\}$ .

Note: There are few more properties like symmetric difference operator, prefix operator, substitution which are closed under closure properties of regular language.

---

## 5.4 Summary

---

In this unit you have learnt about the pumping lemma for regular sets, applications of the pumping lemma, and closure properties of regular sets.

- Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular. If  $L$  is regular, it satisfies Pumping Lemma. If  $L$  does not satisfy Pumping Lemma, it is non-regular.
- The pumping lemma is often used to prove that a particular language is non-regular: a proof by contradiction (of the language's regularity) may consist of exhibiting a word (of the required length) in the language that lacks the property outlined in the pumping lemma.
- Closure properties on regular languages are defined as certain operations on regular language which are guaranteed to produce regular language.

Consider  $L$  and  $M$  are regular languages:

- Kleen Closure
- Positive closure
- Complement
- Reverse Operator
- Complement
- Union
- Intersection
- Set Difference operator
- Inverse Homomorphism

---

## 5.5 Review Questions

---

- Q1. How pumping lemma can be applied to prove that certain sets are not regular?
- Q2. Prove that the language of palindromes over  $\{0, 1\}$  is not regular.
- Q3. Prove that the language containing strings of balanced parentheses is not regular.
- Q4. Prove that Language  $L = \{a^n b^n \text{ for } n \geq 0\}$  is not regular.
- Q5. How do you use pumping lemma for context free languages? Explain with suitable example.



Uttar Pradesh Rajarshi Tandon  
Open University

# **MCS - 113**

## **Master of Computer Science**

### **Theory of Computation**

# **Block**

# **3**

## **Context Free Grammar**

---

### **Unit - 6**

<b>Context Free Grammar</b>	<b>100</b>
-----------------------------	------------

---

### **Unit - 7**

<b>Normal Forms</b>	<b>110</b>
---------------------	------------

---

### **Unit - 8**

<b>Context Free Languages (CFL)</b>	<b>123</b>
-------------------------------------	------------

---

---

## Course Design Committee

---

<b>Prof. Ashutosh Gupta</b> Director (In-charge) School of Computer and Information Sciences, UPRTOU Prayagraj	<b>Chairman</b>
<b>Prof. R. S. Yadav</b> Department of Computer Science and Engineering MNNIT Prayagraj	<b>Member</b>
<b>Dr. Marisha</b> Assistant Professor (Computer Science), School of Sciences, UPRTOU Prayagraj	<b>Member</b>
<b>Mr. Manoj Kumar Balwant</b> Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	<b>Member</b>

---

## Course Preparation Committee

---

<b>Dr. Ravi Shankar Shukla</b> Associate Professor Department of CSE, Invertis University Bareilly-243006, Uttar Pradesh	<b>Author</b>
<b>Prof. Abhay Saxena</b> Professor and Head, Department of Computer Science Dev Sanskriti Vishwavidyalaya, Hardwar, Uttrakhand	<b>Editor</b>
<b>Prof. Ashutosh Gupta</b> Director (In-charge) School of Computer and information, Sciences, UPRTOU Prayagraj	<b>Course Coordinator</b>
<b>Mr. Manoj Kumar Balwant</b> Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	

---

© UPRTOU , Prayagraj - 2023

© MCS - 113 Theory of Computation

ISBN :

---

All Rights are reserved. No Part of this work may reproduced in any form, by mimeograph or any other means, without permission in writing from the Uttar Pradesh Rajarshi Tandon Open University.

Printed and Published by Vinay Kumar, Registrar, Uttar Pradesh rajarshi Tandon Open University, Prayagraj - 2023

**Printed By. – M/s K.C.Printing & Allied Works, Panchwati, Mathura -281003.**



---

## Block- Introductions

---

This is the third block on Theory of computation and having detail description of context free grammar. A Context Free Grammar is a set of rules that define a language. Here, we would like to draw a distinction between Context Free Grammars and grammars for natural languages like English.

Context Free Grammars or CFGs define a formal language. Formal languages work strictly under the defined rules and their sentences are not influenced by the context. And that's where it gets the name context free.

Languages such as English fall under the category of Informal Languages since they are affected by context. They have many other features which a CFG cannot describe.

Even though CFGs cannot describe the context in the natural languages, they can still define the syntax and structure of sentences in these languages. In fact, that is the reason why the CFGs were introduced in the first place.

So we will begin the first unit on context free grammar itself. In this unit firstly we discussed about grammar. A grammar is a set of rules for putting strings together and so corresponds to a language. If we talk about context free grammar, it is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

Second unit begins with Normal forms. In this unit you will know all about simplifications of context free grammar, removal of useless symbols, and removal of epsilon and unit production. You will also learn about Chomsky Normal Form and Grammar to Greibach Normal Form.

In the third unit, we provide another important topic i.e. context free language. In the context free language, we described about closure properties of context free grammar, decision properties of context free language, and applications of context free grammar. We have also discussed about pumping lemma for context free language.

As you study the material, you will find that figures, tables are properly used and these will help to understand the concept. There are many sections in the units to easily understand the topic. Every unit has summary and review questions in the end of the unit which will help you to review yourself.

In your study, you will find that every unit has different equal length and your study time will vary for each unit.

We hope you enjoy studying the material and once again wish you all the best for your success.

---

## UNIT-6 Context-free Grammar

---

### Structure

- 6.0 Introduction
- 6.1 Context Free Grammar (CFG)-Formal definition
- 6.2 Sentential forms
- 6.3 Derivations
  - 6.3.1. Leftmost Derivations
  - 6.3.2. Rightmost Derivations
- 6.4 The language of CFG
- 6.5 Summary
- 6.6 Review Questions

### 6.0 Introduction

This is the first unit of this block. This unit has six sections. Each section has detail knowledge of their topics. Section 1.1 has Context Free Grammar (CFG) and its Formal definition. Section 1.2 describe sentential form. Next section i.e. Section 1.3 define derivation. It has two sub section, explaining left most and right most derivations. In the Section 1.4, you will learn about the language of CFG. Summary is given in the Section 1.5 and Review questions has asked in the section 1.6

### Objective

After studying this unit, you should be able to define:

- Formal definition of Context Free Grammar (CFG)
- Sentential forms
- Leftmost and rightmost derivations
- The language of CFG.

## 6.1 Context Free Grammar (CFG):

Context free grammar: **Context-free grammars** (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe *all* possible languages.

Context-free grammars most commonly used in theoretical computer science, compiler design, and linguistics. CFG's are used to define programming languages and parser programs in compilers and can be generated automatically from context-free grammars.

Context-free grammars can generate context-free languages. They do this by taking a set of variables which are defined recursively, in terms of one another, by a set of production rules. Context-free grammars are named as such because any of the production rules in the grammar can be applied regardless of context—it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it.

### Formal Definition:

A context-free grammar can be described by a four-element tuple  $(V, \Sigma, R, S)$ , where

- $V$  is a finite set of variables (which are non-terminal);
- $\Sigma$  is a finite set (disjoint from  $V$ ) of terminal symbols;
- $R$  is a set of production rules where each production rule maps a variable to a string  $s \in (V \cup \Sigma)^*$ ;
- $S$  (which is in  $V$ ) which is a start symbol.

### Example

- The grammar  $(\{A\}, \{a, b, c\}, P, A)$ ,  $P: A \rightarrow aA, A \rightarrow abc$ .
- The grammar  $(\{S, a, b\}, \{a, b\}, P, S)$ ,  $P: S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \epsilon$
- The grammar  $(\{S, F\}, \{0, 1\}, P, S)$ ,  $P: S \rightarrow 00S \mid 11F, F \rightarrow 00F \mid \epsilon$

### A CFG for Arithmetic Expressions

An example grammar that generates strings representing arithmetic expressions with the four operators  $+$ ,  $-$ ,  $*$ ,  $/$ , and numbers as operands is:

$\langle \text{expression} \rangle \rightarrow \text{number}$

$\langle \text{expression} \rangle \rightarrow (\langle \text{expression} \rangle)$

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle + \langle \text{expression} \rangle$

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle - \langle \text{expression} \rangle$

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle * \langle \text{expression} \rangle$

$\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle / \langle \text{expression} \rangle$

The only non terminal symbol in this grammar is  $\langle \text{expression} \rangle$ , which is also the start symbol. The terminal symbols are  $\{+, -, *, /, (, ), \text{number}\}$ . (We will interpret "number" to represent any valid number.)

The first rule (or production) states that an  $\langle \text{expression} \rangle$  can be rewritten as (or replaced by) a number. In other words, a number is a valid expression.

According to the second rule, an  $\langle \text{expression} \rangle$  enclosed in parentheses is also an  $\langle \text{expression} \rangle$ . One important point should be noted here that this rule describes an expression in terms of expressions, an example of the use of recursion in the definition of context-free grammars.

The remaining rules say that the sum, difference, product, or division of two  $\langle \text{expression} \rangle$ s is also an expression.

### Generating Strings from a CFG

In our grammar for arithmetic expressions, the start symbol is  $\langle \text{expression} \rangle$ , so our initial string is:

$\langle \text{expression} \rangle$

Using rule 5 we can choose to replace this non terminal, producing the string:

$\langle \text{expression} \rangle * \langle \text{expression} \rangle$

We now have two non-terminals to replace. We can apply rule 3 to the first non terminal, producing the string:

$\langle \text{expression} \rangle + \langle \text{expression} \rangle * \langle \text{expression} \rangle$

We can apply rule two to the first non terminal in this string to produce:

$(\langle \text{expression} \rangle) + \langle \text{expression} \rangle * \langle \text{expression} \rangle$

If we apply rule 1 to the remaining non-terminals (the recursion must end somewhere!), we get:

$(\text{number}) + \text{number} * \text{number}$

This is a valid arithmetic expression, as generated by the grammar.

When applying the rules above, we often face a choice as to which production to choose. Different choices will typically result in different strings being generated.

Given a grammar  $G$  with start symbol  $S$ , if there is some sequence of productions that, when applied to the initial string  $S$ , result in the string  $s$ , then  $s$  is in  $L(G)$ , the language of the grammar.

## CFG Examples

A CFG describing strings of letters with the word "main" somewhere in the string:

$\langle \text{program} \rangle \rightarrow \langle \text{letter}^* \rangle \text{ m a i n } \langle \text{letter}^* \rangle$

$\langle \text{letter}^* \rangle \rightarrow \langle \text{letter} \rangle \langle \text{letter}^* \rangle \mid \epsilon$

$\langle \text{letter} \rangle \rightarrow \text{A} \mid \text{B} \mid \dots \mid \text{Z} \mid \text{a} \mid \text{b} \dots \mid \text{z}$

### A CFG for the set of identifiers in Pascal:

$\langle \text{id} \rangle \rightarrow \langle \text{L} \rangle \langle \text{LorD}^* \rangle$

$\langle \text{LorD}^* \rangle \rightarrow \langle \text{L} \rangle \langle \text{LorD}^* \rangle \mid \langle \text{D} \rangle \langle \text{LorD}^* \rangle \mid \epsilon$

$\langle \text{L} \rangle \rightarrow \text{A} \mid \text{B} \mid \dots \mid \text{Z} \mid \text{a} \mid \text{b} \dots \mid \text{z}$

$\langle \text{D} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

### A CFG describing real numbers in Pascal:

$\langle \text{real} \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit}^* \rangle \langle \text{decimal part} \rangle \langle \text{exp} \rangle$

$\langle \text{digit}^* \rangle \rightarrow \langle \text{digit} \rangle \langle \text{digit}^* \rangle \mid \epsilon$

$\langle \text{decimal part} \rangle \rightarrow \text{'.'} \langle \text{digit} \rangle \langle \text{digit}^* \rangle \mid \epsilon$

$\langle \text{exp} \rangle \rightarrow \text{'E'} \langle \text{sign} \rangle \langle \text{digit} \rangle \langle \text{digit}^* \rangle \mid \epsilon$

$\langle \text{sign} \rangle \rightarrow + \mid - \mid \epsilon$

$\langle \text{digit} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

### A CFG for C++ compound statements:

$\langle \text{compound stmt} \rangle \rightarrow \{ \langle \text{stmt list} \rangle \}$

$\langle \text{stmt list} \rangle \rightarrow \langle \text{stmt} \rangle \langle \text{stmt list} \rangle \mid \epsilon$

$\langle \text{stmt} \rangle \rightarrow \langle \text{compound stmt} \rangle$

$\langle \text{stmt} \rangle \rightarrow \text{if}(\langle \text{expr} \rangle) \langle \text{stmt} \rangle$

$\langle \text{stmt} \rangle \rightarrow \text{if}(\langle \text{expr} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

$\langle \text{stmt} \rangle \rightarrow \text{while} (\langle \text{expr} \rangle) \langle \text{stmt} \rangle$   
 $\langle \text{stmt} \rangle \rightarrow \text{do } \langle \text{stmt} \rangle \text{ while } (\langle \text{expr} \rangle);$   
 $\langle \text{stmt} \rangle \rightarrow \text{for} (\langle \text{stmt} \rangle \langle \text{expr} \rangle; \langle \text{expr} \rangle) \langle \text{stmt} \rangle$   
 $\langle \text{stmt} \rangle \rightarrow \text{case } \langle \text{expr} \rangle: \langle \text{stmt} \rangle$   
 $\langle \text{stmt} \rangle \rightarrow \text{switch} (\langle \text{expr} \rangle) \langle \text{stmt} \rangle$   
 $\langle \text{stmt} \rangle \rightarrow \text{break}; \mid \text{continue};$   
 $\langle \text{stmt} \rangle \rightarrow \text{return } \langle \text{expr} \rangle; \mid \text{goto } \langle \text{id} \rangle;$

### Finding all the Strings Generated by a CFG

There are several ways to generate the (possibly infinite) set of strings generated by a grammar. We will show a technique based on the number of productions used to generate the string.

Find the strings generated by the following CFG:

$\langle S \rangle \rightarrow w c d \langle S \rangle \mid b \langle L \rangle e \mid s$   
 $\langle L \rangle \rightarrow \langle L \rangle ; \langle S \rangle \mid \langle S \rangle$

0. Applying at most zero productions, we cannot generate any strings.

1. Applying at most one production (starting with the start symbol). With the help of this we can generate  $\{wcd\langle S \rangle, b\langle L \rangle e, s\}$ . Only one of these strings contains entirely of terminal symbols, so the set of terminal strings we can generate using at most one production is  $\{s\}$ .

2. Applying at most two productions, we can generate all the strings we can generate with one production, plus any additional strings we can generate with an additional production.

$\{wcdwcd\langle S \rangle, wcd b\langle L \rangle e, wcds, b\langle S \rangle e, b\langle L \rangle ; \langle S \rangle e, s\}$

The set of terminal strings we can generate with at most two productions is therefore  $\{s, wcds\}$ .

3. Applying at most three productions, we can generate:

$\{wcdwcdwcd\langle S \rangle, wcdwcd b\langle L \rangle e, wcdwcds, wcd b\langle L \rangle ; \langle S \rangle e,$   
 $wcd b\langle S \rangle e, bwcd\langle S \rangle e, bb\langle L \rangle ee, bse, b\langle L \rangle ; \langle S \rangle Se,$   
 $b\langle S \rangle \langle S \rangle e, b\langle L \rangle wcd\langle S \rangle e, b\langle L \rangle b\langle L \rangle ee, b\langle L \rangle Se\}$

The set of terminal strings we can generate with at most three productions is therefore  $\{s, wcds, wcdwcds, bse\}$ .

We can repeat this process for an arbitrary number of steps  $N$ , and find all the strings the grammar can generate by applying  $N$  productions.

## 6.2 Sentential Forms

A *sentential form* is the start symbol  $S$  of a grammar or any string in  $(V \cup T)^*$  that can be derived from  $S$ .

For example: Consider the linear grammar

$(\{S, B\}, \{a, b\}, S, \{S \rightarrow aS, S \rightarrow B, B \rightarrow bB, B \rightarrow \lambda\})$ .

A derivation using this grammar might look like this:

$S \Rightarrow aS \Rightarrow aB \Rightarrow abB \Rightarrow abbB \Rightarrow abb$

Each of  $\{S, aS, aB, abB, abbB, abb\}$  is a sentential form.

Because this grammar is linear, each sentential form has at most one variable. Hence there is never any choice about which variable to expand next.

## 6.3 Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing, we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be placed with production rule.

### 6.3.1. Leftmost Derivation:

According to the leftmost derivation, the input in the production rule can be scanned and replaced from left to right. So in leftmost derivation, we read the input string from left to right.

**Example:**

**Production rules:**

1.  $E = E + E$
2.  $E = E - E$
3.  $E = a \mid b$

## Input

1.  $a - b + a$

### The leftmost derivation is:

1.  $E = E + E$
2.  $E = E - E + E$
3.  $E = a - E + E$
4.  $E = a - b + E$
5.  $E = a - b + a$

### 1.3.2. Rightmost Derivation:

In rightmost derivation, the input is scanned and replaced with the production rule from right to left. So in rightmost derivation, we read the input string from right to left.

#### Example 1:

##### Production rules:

1.  $E = E + E$
2.  $E = E - E$
3.  $E = a \mid b$

## Input

1.  $a - b + a$

### The rightmost derivation is:

1.  $E = E - E$
2.  $E = E - E + E$
3.  $E = E - E + a$
4.  $E = E - b + a$
5.  $E = a - b + a$

When we use the leftmost derivation or rightmost derivation, we may get the same string. This type of derivation does not effect on getting of a string.

#### Example 2:

Derive the string "aabbabba" for leftmost derivation and rightmost derivation using a CFG given by,

1.  $S \rightarrow aB \mid bA$
2.  $S \rightarrow a \mid aS \mid bAA$
3.  $S \rightarrow b \mid aS \mid aBB$



**Solution:****Leftmost derivation:**

1. S
2. aB       $S \rightarrow aB$
3. aaBB     $B \rightarrow aBB$
4. aabB     $B \rightarrow b$
5. aabbS     $B \rightarrow bS$
6. aabbaB    $S \rightarrow aB$
7. aabbabS    $B \rightarrow bS$
8. aabbabbA    $S \rightarrow bA$
9. aabbabba    $A \rightarrow a$

**Rightmost derivation:**

1. S
2. aB       $S \rightarrow aB$
3. aaBB     $B \rightarrow aBB$
4. aaBbS     $B \rightarrow bS$
5. aaBbbA     $S \rightarrow bA$
6. aaBbba     $A \rightarrow a$
7. aabSbba     $B \rightarrow bS$
8. aabbAbba    $S \rightarrow bA$
9. aabbabba    $A \rightarrow a$

**Example 3:**

Derive the string "00101" for leftmost derivation and rightmost derivation using a CFG given by,

1.  $S \rightarrow A1B$
2.  $A \rightarrow 0A \mid \varepsilon$
3.  $B \rightarrow 0B \mid 1B \mid \varepsilon$

**Solution:****Leftmost derivation:**

1. S
2. A1B
3. 0A1B
4. 00A1B
5. 001B
6. 0010B
7. 00101B
8. 00101

**Rightmost derivation:**

1. S
2. A1B
3. A10B
4. A101B
5. A101
6. 0A101
7. 00A101
8. 00101

---

## Check your progress

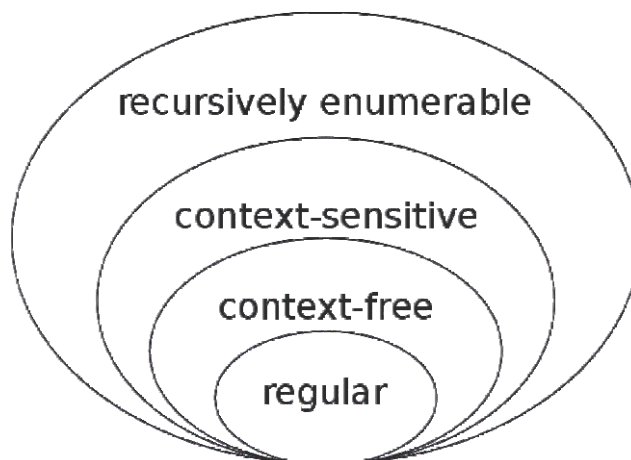
---

- Q1. What is context-free grammar? Explain with suitable example.
- Q2. Define sensational form in CFG.
- Q3. For the string 00110101, find the leftmost derivation, right most derivation.

1.  $S \rightarrow 0B \mid 1A,$
2.  $A \rightarrow 0 \mid 0S \mid 1AA,$
3.  $B \rightarrow 1 \mid 1S \mid 0BB$

## 6.4 The Language of CFG

In the theory of automata, context-free grammars generate Context-free languages (CFLs). The set of all context-free languages is same as the set of languages accepted by pushdown automata, and the set of regular languages is a subset of context-free languages. A computational model accepts an inputted language if it runs with the help of the model and ends in an accepting final state. One important point must be noted here that all regular languages are context-free languages, but not all context-free languages are regular. Most arithmetic expressions are generated by context-free grammars, and are therefore, context-free languages. Context-free languages and context-free grammars have applications in computer science and linguistics such as natural language processing and computer language design.



**Fig 1: Language Design**

---

## 6.5 Summary

---

In this unit you have learnt about Context Free Grammar and its properties. You have also learnt about sentential forms, derivations of left most and right most, and the language of context free grammar.

- **Context Free Grammars or CFGs** define a formal language. Formal languages work strictly under the defined rules and their sentences are not influenced by the context.
- Every string of symbols in the derivation is a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded next in the derivation
- A rightmost derivation works right to left instead
- Some derivations are neither leftmost nor rightmost

---

## 6.6 Review Questions

---

Q1. What is the use of context-free grammar in the theory of automata?

Q2. Why do we use sentential forms in derivation?

Q3. Derive the string "00101" for leftmost derivation and rightmost derivation using a CFG given by,

$$S \rightarrow A1B$$

$$A \rightarrow 0A \mid \epsilon$$

$$B \rightarrow 0B \mid 1B \mid \epsilon$$

Q4. Prove that Context-free languages are closed under the union operation.

Q5. Use the Pumping Lemma to prove that  $L = \{a^n b^n c^n \mid n > 0\}$  is not a context-free language.

---

## UNIT-7 Normal Forms

---

### Structure

#### 7.0 Introduction

#### 7.1 Simplifications of CFG's

##### 7.1.1 Removal of Useless Symbols

##### 7.1.2 Removal of epsilon

##### 7.1.3 Removal of Unit Production

#### 7.2 Normal Forms

##### 7.2.1 CNF

##### 7.2.2 GNF.

#### 7.3 Summary

#### 7.4 Review Questions

### 7.0 Introduction

This is the second unit of this block. There are four sections in this unit. There are some sub-sections of these sections. In the section 2.1, you will learn about Simplifications of CFG's. This section also has three sub-sections. Removal of Useless Symbols, Removal of epsilon and Unit Production has been defined in these sections. In the section 2.2, you will get normal forms of context free grammar. Under this section, there are two sub-sections, i.e. Section 2.2.1 and 2.2.2. In these sections you will learn about CNF and GNF. Last two sections provide summary and review questions.

## Objective

After studying this unit, you should be able to define:

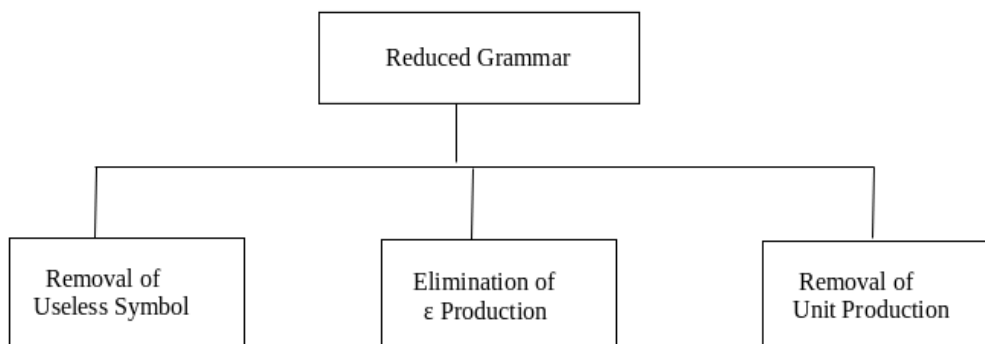
- Simplifications of CFG's i.e. Removal of Useless Symbols, Removal of epsilon and Unit Production
- Normal Forms-CNF and GNF.

## 7.1 Simplifications of CFG

As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammars are not always optimized that means the grammar may consist of some extra symbols (non-terminal). Having extra symbols, unnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below:

- Each variable (i.e. non-terminal) and each terminal of  $G$  appears in the derivation of some word in  $L$ .
- There should not be any production as  $X \rightarrow Y$  where  $X$  and  $Y$  are non-terminal.
- If  $\epsilon$  is not in the language  $L$ , then there need not to be the production  $X \rightarrow \epsilon$ .

Let us study the reduction process in detail.



**Fig: 2.1 Simplification diagram of CFG**

### 7.1.1 Removal of Useless Symbols

A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it

does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

$$T \rightarrow aaB \mid abA \mid aaT$$

$$A \rightarrow aA$$

$$B \rightarrow ab \mid b$$

$$C \rightarrow ad$$

In the above example, the variable 'C' will never occur in the derivation of any string, so the production  $C \rightarrow ad$  is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production  $A \rightarrow aA$  is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production  $A \rightarrow aA$ , we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

## 7.1.2 Removal of Epsilon

The productions  $S \rightarrow \varepsilon$  are called  $\varepsilon$  productions. The production can only be removed from those grammars that do not generate  $\varepsilon$ .

Step 1: First find out all null able non-terminal variable which derives  $\varepsilon$ .

Step 2: For each production  $A \rightarrow a$ , construct all production  $A \rightarrow x$ , where x is obtained from a by removing one or more non-terminal from step 1.

Step 3: Now combine the result of step 2 with the original production and remove  $\varepsilon$  productions.

**Example:**

Remove the production from the following CFG by preserving the meaning of it.

$$S \rightarrow XYX$$

$$X \rightarrow 0X \mid \varepsilon$$

$$Y \rightarrow 1Y \mid \varepsilon$$

**Solution:**

Now, while removing  $\varepsilon$  production, we are deleting the rule  $X \rightarrow \varepsilon$  and  $Y \rightarrow \varepsilon$ . To preserve the meaning of CFG we are actually placing  $\varepsilon$  at the right-hand side whenever X and Y have appeared.

Let us take

$$S \rightarrow XYX$$

If the first X at right-hand side is  $\varepsilon$ . Then

$$S \rightarrow YX$$

Similarly, if the last X in R.H.S. =  $\varepsilon$ . Then

$$S \rightarrow XY$$

If  $Y = \varepsilon$  then

$$S \rightarrow XX$$

If Y and X are  $\varepsilon$  then,

$$S \rightarrow X$$

If both X are replaced by  $\varepsilon$

$$S \rightarrow Y$$

Now,

$$S \rightarrow XY \mid YX \mid XX \mid X \mid Y$$

Now let us consider

$$X \rightarrow 0X$$

If we place  $\varepsilon$  at right-hand side for X then,

$$X \rightarrow 0$$

$$X \rightarrow 0X \mid 0$$

Similarly,  $Y \rightarrow 1Y \mid 1$

Collectively we can rewrite the CFG with removed  $\varepsilon$  production as

$$S \rightarrow XY \mid YX \mid XX \mid X \mid Y$$

$$X \rightarrow 0X \mid 0$$

$$Y \rightarrow 1Y \mid 1$$

### 7.1.3 Removal of Unit Production

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

Step 1: To remove  $X \rightarrow Y$ , add production  $X \rightarrow a$  to the grammar rule whenever  $Y \rightarrow a$  occurs in the grammar.

Step 2: Now delete  $X \rightarrow Y$  from the grammar.

Step 3: Repeat step 1 and step 2 until all unit productions are removed.

**For example:**

$$S \rightarrow 0A \mid 1B \mid C$$

$$A \rightarrow 0S \mid 00$$

$$B \rightarrow 1 \mid A$$

$$C \rightarrow 01$$

**Solution:**

$S \rightarrow C$  is a unit production. But while removing  $S \rightarrow C$  we have to consider what  $C$  gives. So, we can add a rule to  $S$ .

$$S \rightarrow 0A \mid 1B \mid 01$$

Similarly,  $B \rightarrow A$  is also a unit production so we can modify it as

$$B \rightarrow 1 \mid 0S \mid 00$$

Thus finally we can write CFG without unit production as

$$S \rightarrow 0A \mid 1B \mid 01$$

$$A \rightarrow 0S \mid 00$$

$$B \rightarrow 1 \mid 0S \mid 00$$

$$C \rightarrow 01$$



---

## Check your progress

---

Q1. What is the role of simplifications of CFG in the theory of automata?

Q2. What is the use of removal of useless symbols?

Q3.  $S \rightarrow AB$ ,  
 $A \rightarrow a$ ,  
 $B \rightarrow C \mid b$ ,  
 $C \rightarrow D$ ,  
 $D \rightarrow E$  and  
 $E \rightarrow a$ .

Eliminate unit productions and get an equivalent grammar.

## 7.2 Normal Forms

A normal form  $F$  for a set  $C$  of data objects is a form, i.e., a set of syntactically valid objects, with the following two properties:

- For every element  $c$  of  $C$ , except possibly a finite set of special cases, there exists some element  $f$  of  $F$  such that  $f$  is equivalent to  $c$  with respect to some set of tasks.
- $F$  is simpler than the original form in which the elements of  $C$  are written. By “simpler” we mean that at least some tasks are easier to perform on elements of  $F$  than they would be on elements of  $C$ .

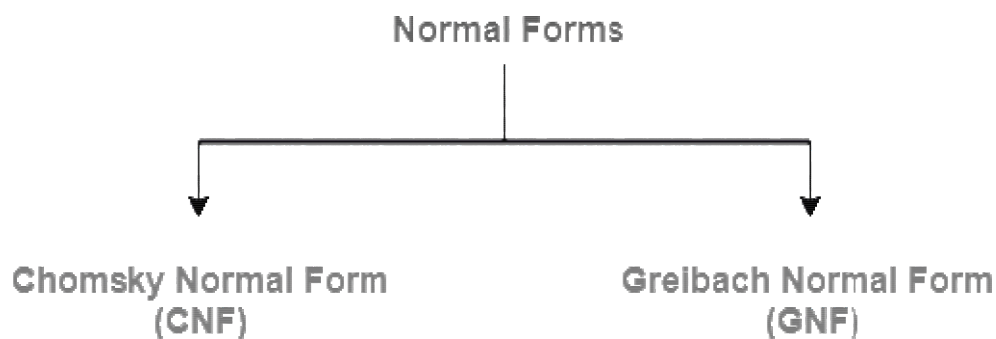


Fig 7.1: Normal Forms classification

### 7.2.1 CNF (Chomsky Normal Form)

CNF stands for Chomsky normal form. A CFG (context free grammar) is in CNF (Chomsky normal form) if all production rules satisfy one of the following conditions:

Start symbol generating  $\epsilon$ . For example,  $A \rightarrow \epsilon$ .

A non-terminal generating two non-terminals. For example,  $S \rightarrow AB$ .

A non-terminal generating a terminal. For example,  $S \rightarrow a$ .

For example:

$$G1 = \{S \rightarrow AB, S \rightarrow c, A \rightarrow a, B \rightarrow b\}$$

$$G2 = \{S \rightarrow aA, A \rightarrow a, B \rightarrow c\}$$

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as  $S \rightarrow aZ$  contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

### Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

$$S1 \rightarrow S$$

Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production  $S \rightarrow aA$  can be decomposed as:

$$S \rightarrow RA$$

$$R \rightarrow a$$

Step 4: Eliminate RHS with more than two non-terminals. For example,  $S \rightarrow ASB$  can be decomposed as:

$$S \rightarrow RS$$

$$R \rightarrow AS$$

Example:

Convert the given CFG to CNF. Consider the given grammar G1:

$$S \rightarrow a \mid aA \mid B$$

$$A \rightarrow aBB \mid \varepsilon$$

$$B \rightarrow Aa \mid b$$

**Solution:**

Step 1: We will create a new production  $S1 \rightarrow S$ , as the start symbol S appears on the RHS. The grammar will be:

$$S1 \rightarrow S$$

$$S \rightarrow a \mid aA \mid B$$

$$A \rightarrow aBB \mid \varepsilon$$

$$B \rightarrow Aa \mid b$$

Step 2: As grammar G1 contains  $A \rightarrow \varepsilon$  null production, its removal from the grammar yields:

$$S1 \rightarrow S$$

$$S \rightarrow a \mid aA \mid B$$

$$A \rightarrow aBB$$

$$B \rightarrow Aa \mid b \mid a$$

Now, as grammar G1 contains Unit Production  $S \rightarrow B$ , its removal yield:

$$S1 \rightarrow S$$

$$S \rightarrow a \mid aA \mid Aa \mid b$$

$$A \rightarrow aBB$$

$$B \rightarrow Aa \mid b \mid a$$

Also remove the unit production  $S1 \rightarrow S$ , its removal from the grammar yields:

$$S_0 \rightarrow a \mid aA \mid Aa \mid b$$

$$S \rightarrow a \mid aA \mid Aa \mid b$$

$$A \rightarrow aBB$$

$$B \rightarrow Aa \mid b \mid a$$

Step 3: In the production rule  $S_0 \rightarrow aA \mid Aa$ ,  $S \rightarrow aA \mid Aa$ ,  $A \rightarrow aBB$  and  $B \rightarrow Aa$ , terminal  $a$  exists on RHS with non-terminals. So we will replace terminal  $a$  with  $X$ :

$$S_0 \rightarrow a \mid XA \mid AX \mid b$$

$$S \rightarrow a \mid XA \mid AX \mid b$$

$$A \rightarrow XBB$$

$$B \rightarrow AX \mid b \mid a$$

$$X \rightarrow a$$

Step 4: In the production rule  $A \rightarrow XBB$ , RHS has more than two symbols, removing it from grammar yield:

$$S_0 \rightarrow a \mid XA \mid AX \mid b$$

$$S \rightarrow a \mid XA \mid AX \mid b$$

$$A \rightarrow RB$$

$$B \rightarrow AX \mid b \mid a$$

$$X \rightarrow a$$

$$R \rightarrow XB$$

Hence, for the given grammar, this is the required CNF.

## 7.2.2 Greibach Normal Form (GNF)

In the theory of automata, a CFG (context free grammar) is in GNF (Greibach normal form) if there are following conditions satisfy all the production rules. The conditions is given blow:

- A start symbol producing  $\epsilon$ . For example,  $S \rightarrow \epsilon$ .
- A non-terminal producing a terminal. For example,  $A \rightarrow a$ .
- A non-terminal generating a terminal which is followed by any number of non-terminals. For example,  $S \rightarrow aASB$ .

For example:

$$G1 = \{S \rightarrow aAB \mid aB, A \rightarrow aA \mid a, B \rightarrow bB \mid b\}$$

$$G2 = \{S \rightarrow aAB \mid aB, A \rightarrow aA \mid \varepsilon, B \rightarrow bB \mid \varepsilon\}$$

The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF. However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as  $A \rightarrow \varepsilon$  and  $B \rightarrow \varepsilon$  contains  $\varepsilon$  (only start symbol can generate  $\varepsilon$ ). So the grammar G2 is not in GNF.

### Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

Step 2: If the grammar exists left recursion, eliminate it.

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

Step 3: In the grammar, convert the given production rule into GNF form.

If any production rule in the grammar is not in GNF form, convert it.

#### Example:

$$S \rightarrow XB \mid AA$$

$$A \rightarrow a \mid SA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

#### Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.

The production rules  $A \rightarrow SA$  is not in GNF, so we substitute  $S \rightarrow XB \mid AA$  in the production rule  $A \rightarrow SA$  as:

$$S \rightarrow XB \mid AA$$

$$A \rightarrow a \mid XBA \mid AAA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

The production rule  $S \rightarrow XB$  and  $B \rightarrow XBA$  is not in GNF, so we substitute  $X \rightarrow a$  in the production rule  $S \rightarrow XB$  and  $B \rightarrow XBA$  as:

$$S \rightarrow aB \mid AA$$

$$A \rightarrow a \mid aBA \mid AAA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

Now we will remove left recursion ( $A \rightarrow AAA$ ), we get:

$$S \rightarrow aB \mid AA$$

$$A \rightarrow aC \mid aBAC$$

$$C \rightarrow AAC \mid \varepsilon$$

$$B \rightarrow b$$

$$X \rightarrow a$$

Now we will remove null production  $C \rightarrow \varepsilon$ , we get:

$$S \rightarrow aB \mid AA$$

$$A \rightarrow aC \mid aBAC \mid a \mid aBA$$

$$C \rightarrow AAC \mid AA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

The production rule  $S \rightarrow AA$  is not in GNF, so we substitute  $A \rightarrow aC \mid aBAC \mid a \mid aBA$  in production rule  $S \rightarrow AA$  as:

$$S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$$

$$A \rightarrow aC \mid aBAC \mid a \mid aBA$$

$$C \rightarrow AAC$$

$$C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

The production rule  $C \rightarrow AAC$  is not in GNF, so we substitute  $A \rightarrow aC \mid aBAC \mid a \mid aBA$  in production rule  $C \rightarrow AAC$  as:

$$S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$$

$$A \rightarrow aC \mid aBAC \mid a \mid aBA$$

$$C \rightarrow aCAC \mid aBACAC \mid aAC \mid aBAAC$$

$$C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

Hence, this is the GNF form for the grammar G.

---

## 7.3 Summary

---

In this unit you have learnt about Normal Forms of Context Free Grammar. You have also learnt about simplifications of CFG like removal of useless symbols, removal of epsilon and rightmost derivations. You have also learnt about Chomsky normal forms and Greibach normal forms.

- In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions. Elimination of these productions and symbols is called **simplification of CFGs**.
- The productions that can never take part in derivation of any string, are called **useless productions**.
- The productions of type ' $A \rightarrow \lambda$ ' are called  $\lambda$  productions (also called **lambda productions and null productions**). These productions can only be removed from those grammars that do not generate  $\lambda$  (an empty string). It is possible for a grammar to contain null productions and yet not produce an empty string.
- The productions of type ' $A \rightarrow B$ ' are called **unit productions**.

---

## 7.4 Review Questions

---

Q1. What do you understand by simplifications of context-free grammar? Elaborate your answer.

Q2. Remove unit production from the following string:

$$S \rightarrow XY, X \rightarrow a, Y \rightarrow Z \mid b, Z \rightarrow M, M \rightarrow N, N \rightarrow a$$

Q3. Remove null production from the following string:

$$S \rightarrow ASA \mid aB \mid b, A \rightarrow B, B \rightarrow b \mid \epsilon$$

Q4. Convert the following CFG into GNF

$$S \rightarrow XY \mid X_n \mid p$$

$$X \rightarrow mX \mid m$$

$$Y \rightarrow X_n \mid o$$

Q5. Convert the following grammar into Greibach Normal Form (GNF).

$$S \rightarrow XA \mid BB$$

$$B \rightarrow b \mid SB$$

$$X \rightarrow b$$

$$A \rightarrow a$$

Q6. Reduce the following grammars to Chomsky normal form:

$$\begin{aligned} \text{a. } S &\rightarrow 1A \mid 0B, \\ A &\rightarrow 1AA \mid 0S \mid 0, \\ B &\rightarrow 0BB \mid 1S \mid 1 \end{aligned}$$

$$\begin{aligned} \text{b. } S &\rightarrow abSb \mid a \mid aAb, \\ A &\rightarrow bS \mid aAAb \end{aligned}$$

Q7. Reduce the following grammars to Greibach Normal Form:

$$\text{a. } S \rightarrow SS, S \rightarrow 0S1 \mid 01$$

$$\begin{aligned} \text{b. } S &\rightarrow A0, \\ A &\rightarrow 0B, \\ B &\rightarrow A0, \\ B &\rightarrow 1 \end{aligned}$$



---

# UNIT-8 Context Free Languages (CFL)

---

## Structure

- 8.0 Introduction
- 8.1 Context-free Languages
- 8.2 Closure Properties
- 8.3 Decision Properties of CFL
- 8.4 Application of CFG
- 8.5 The Pumping Lemma for Context-Free Languages
- 8.6 Summary
- 8.7 Review Questions

## 8.0 Introduction

This is the third and last unit of this block. In this unit, there are seven sections. In the section 3.1, you will learn about context-free languages. Section 3.2. Closure Properties of CFL has been define. Decision Properties of CFL has been defined in the Section 3.3. You will know all about of application of CFG in the section 3.4. Another section i.e. Section 3.5 defines Pumping Lemma for CFL. Section 3.6 has summary and Section 3.7 has review questions.

## Objective

After studying this unit, you should be able to define:

- Closure Properties of CFL
- Decision Properties of CFL
- Application of CFG
- Pumping Lemma for CFL.

## 8.1 Context-free Languages

In formal language theory, a **language** is defined as a set of strings of symbols that may be constrained by specific rules. Similarly, the written English language is made up of groups of letters (words) separated by spaces. A valid (accepted) sentence in the language must follow particular rules, the grammar.

A context-free language is a language generated by a context-free grammar. They are more general (and include) regular languages. The same context-free language might be generated by multiple context-free grammars.

The set of all context-free languages is identical to the set of languages that are accepted by pushdown automata (PDA).

Here is an example of a language that is not regular but *is* context-free:

$\{a^n b^n | n \geq 0\}$ . This is the language of all strings that have an equal number of a's and b's.

In this notation,  $a^4 b^4$  can be expanded out to aaaabbbb, where there are four a's and then four b's. (So this isn't exponentiation, though the notation is similar).

## 8.2 Closure Properties

Context-free languages have the following closure properties. A set is **closed** under an operation if doing the operation on a given set always produces a member of the same set. This means that if one of these closed operations is applied to a context-free language the result will also be a context-free language.

- **Union:** Context-free languages are closed under the union operation. This means that if L and P are both context-free languages, then  $L \cup P$  is also a context-free language.

### Proof:

Here is a proof that context-free grammars are closed under union.

1. Let L and P be generated by the context-free grammars,  $G_L = (V_L, \Sigma_L, R_L, S_L)$  and  $G_P = (V_P, \Sigma_P, R_P, S_P)$  respectively.
2. Without loss of generality, subscript each nonterminal symbol in  $G_L$  with an L, and each nonterminal of  $G_P$  with a P such that  $V_L \cap V_P = \emptyset$
3. Define the CFG, G, that generates  $L \cup P$  as follows:  
 $G = (V_L \cup V_P \cup \{S\}, \Sigma_L \cup \Sigma_P, R_L \cup R_P \cup \{S \rightarrow S_L | S_P\}, S)$

- **Concatenation:** If L and P are both context-free languages, then  $LP$  is also context free. The concatenation of a string is defined as follows:  $S_1 S_2 = vw : v \in S_1 \text{ and } w \in S_2$ .

### Proof:

Here is a proof that context-free grammars are closed under concatenation. This proof is similar to the union closure proof.

1. Let  $P$  be generated by the context-free grammars,  $G_L=(V_L,\Sigma_L,R_L,S_L)$  and  $G_P=(V_P,\Sigma_P,R_P,S_P)$  respectively.
2. Without loss of generality, subscript each nonterminal symbol in  $G_L$  with an  $L$ , and each nonterminal of  $G_P$  a  $P$  such that  $V_L \cap V_P = \emptyset$ .
3. Define the CFG,  $G$ , that generates  $L \cup P$  as follows:  
 $G=(V_L \cup V_P \cup \{S\}, \Sigma_L \cup \Sigma_P, R_L \cup R_P \cup \{S \rightarrow S_L S_P\}, S)$ .

Every word that  $G$  generates is a word in  $L$  followed by a word in  $P$ , which is the definition of concatenation.

- **Kleene Star:** If  $L$  is a context-free language, then  $L^*$  is also context free. The Kleene star can repeat the string or symbol it is attached to any number of times (including zero times). The Kleene star basically performs a recursive concatenation of a string with itself. For example,  $\{a,b\}^* = \{\epsilon, a, b, ab, aab, aaab, abb, \dots\}$  and so on. We've already proved that CFLs are closed under concatenation.

Context-free languages are **not** closed under complement or intersection.

If CFL's were closed under intersection, then there would be CFLs that violate the pumping lemma for context-free languages (see the next section for more details) which cannot be.

### Proof:

Take two context-free languages  $L = \{a^n b^n c^m\}$  and  $P = \{a^n b^m c^m\}$ . The intersection of  $L$  and  $P$ ,  $L \cap P = \{a^n b^n c^n\}$ , which we will see below in the pumping lemma for context-free languages, is not a context-free language.

## 8.3 Decision Properties of CFL

In the theory of automata, there are many decision problems are solvable for simple machine models, such as finite automata or pushdown automata if we talk about deterministic finite automata, problems similar to equivalence can be resolved even in polynomial time. Also there are effective parsing algorithms for context-free grammars.

As we know that the following important characterization:

Regular languages = languages denoted by regular expressions

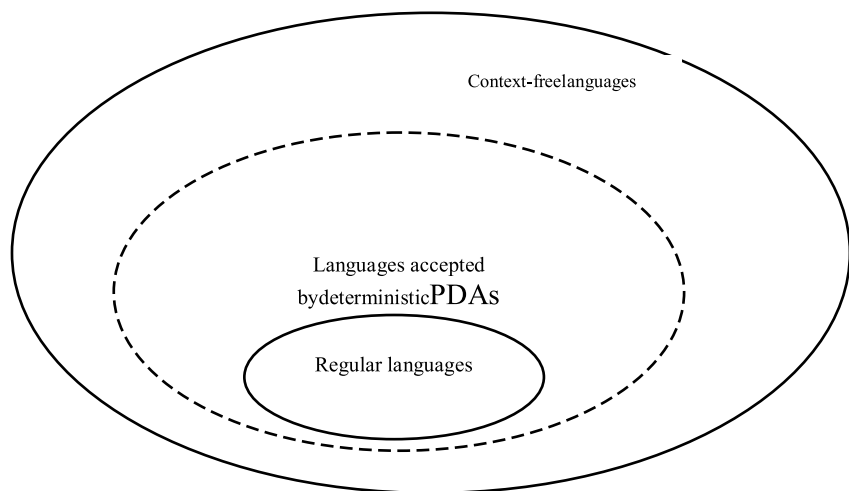
= languages accepted by DFAs (deterministic finite automata)

= languages accepted by NFAs (nondeterministic finite automata).

The class of regular languages is firmly confined in the deterministic context-free languages (DCFL) which in turn are strictly contained in the (general) context-free languages. The class DCFL consists of languages recognized by deterministic pushdown automata.

We recall the following basic notions. A decision problem is a restricted type of an algorithmic problem where for each input there are only two possible outputs.

- A decision problem is a function that acquaintances with each input instance of the problem a truth value true or false.
- A decision algorithm is an algorithm that computes the correct truth value for each input instance of a decision problem. The algorithm has to terminate on all inputs!
- A decision problem is decidable if there exists a decision algorithm for it. Otherwise it is undecidable.



**Figure 8.1: Regular, context-free and deterministic context-free languages**

To prove that a decision problem is decidable it is sufficient to give an algorithm for it. On the other hand, the question is arising that how could we possibly establish (= prove) that some decision problem is undecidable?

### **Decidability properties of regular languages**

There are some important decision problems for finite automata include the following:

## 1. Membership of DFA

Instance: A DFA  $M = (Q, \Sigma, \delta, q_0, F)$  and a string  $w \in \Sigma^*$

Question: Is  $w \in L(M)$ ?

Proposition. DFA membership is decidable.

Proof. To be explained in class: the algorithm simulates the given DFA on the given input.

## 2. DFA emptiness

Instance: A DFA  $M = (Q, \Sigma, \delta, q_0, F)$

Question: Is  $L(M) = \emptyset$ ?

Theorem. DFA emptiness is decidable.

Proof. We note that  $L(M) = \emptyset$  iff there is no path in the state diagram of  $M$  from  $q_0$  to a final state. If  $F = \emptyset$ , then clearly  $L(M) = \emptyset$ . Otherwise, we use a graph reachability algorithm to enumerate all states that can be reached from  $q_0$  and check whether this set contains some state of  $F$ . The algorithm terminates because the state diagram is finite.

## 3. DFA universality

Instance: A DFA  $M = (Q, \Sigma, \delta, q_0, F)$

Question: Is  $L(M) = \Sigma^*$ ?

Theorem. DFA universality is decidable.

## 4. DFA containment

Instance: Two DFAs  $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$  and  $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$

Question: Is  $L(M_1) \subseteq L(M_2)$ ?

Theorem. DFA containment is decidable.

## 5. DFA equivalence

Instance: Two DFAs  $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$  and  $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$

Question: Is  $L(M_1) = L(M_2)$ ?

Theorem. DFA equivalence is decidable.

Regular languages are useful for many practical applications due to the fact that all natural questions concerning regular languages are decidable. The downside is that the family of regular languages is quite small.

---

## Check your progress

---

Q1. What is the role of Context Free language in the theory of automata?

Q2. Define Kleene Star with proof.

### 8.4 Applications of Context-free grammar

Grammars are used to describe programming languages. Most importantly, there is a mechanical way of turning the description as a Context Free Grammar (CFG) into a parser, the component of the compiler that discovers the structure of the source program and represents that structure as a tree.

For example, The Document Type Definition (DTD) feature of XML (Extensible Mark-up Language) is essentially a context-free grammar that describes the allowable HTML tags and the ways in which these tags may be nested. For example, one could describe a sequence of characters that was intended to be interpreted as a phone number by `<PHONE>` and `</PHONE>`

#### Example-1:

Typical programming languages use parentheses and or brackets in a nested and balanced fashion. That is, we must be able to match some left parenthesis against a right parenthesis that appears immediately to its right, remove both of them and repeat. If we eventually eliminate all the parenthesis, then the string will be balanced. Example of strings with balanced parenthesis are `()`, `()()`, `(( ))`, while `(`, `)` and `((` are not balanced. A grammar with the following productions generates all and only the strings with balanced parenthesis:

$B \rightarrow BB \mid (B) \mid \lambda$

The first production,  $B \rightarrow BB$ , says that concatenation of two strings of balanced parenthesis is balanced. That is, we can match the parenthesis in two strings independently. The second production,  $B \rightarrow (B)$ , says that if we place a pair of parenthesis around a balanced string, then the result is balanced. The third production,  $B \rightarrow \lambda$  is the basis, which says that an empty string is balanced.

#### Example-2:

There are numerous aspects of typical programming language that have like balanced parentheses. Beginning and ending of code blocks, such as `begin` and `end` in Pascal, or the curly braces `{ . . }` of C, are examples. There is a related pattern that appears occasionally, where “parentheses” can be balanced with the exception that there can be unbalanced left parentheses. An example is the treatment of *if* and *else* in C. An if-clause can appear unbalanced by any else-clause, or it may be balanced by a matching else-clause. A grammar that generates the possible sequence of *if* and *else* (represented by *i* and *e*, respectively) is:

$$S \rightarrow SS|iS|iSe|\lambda$$

For instance, ieie, iie, and iei are possible sequences of *if* and *else*'s and each of these strings is generated by the above grammar. Some examples of illegal sequences not generated by the grammar are ei, ieeii, iee.

### Example-3:

We give below CFG that describes some parts of the structure of HTML (Hypertext Mark-up Language).

$$\text{Char} \rightarrow a|A|. \dots$$

$$\text{Text} \rightarrow \lambda|\text{Char Text}$$

$$\text{Doc} \rightarrow \lambda|\text{Element Doc}$$

$$\text{Element} \rightarrow \text{Text} | \langle \text{EM} \rangle \text{Doc} \langle / \text{EM} \rangle | \langle \text{P} \rangle \text{Doc} | \langle \text{OL} \rangle \text{List} \langle / \text{OL} \rangle$$

$$\text{List} \rightarrow \lambda | \text{ListItem List}$$

$$\text{ListItem} \rightarrow \langle \text{LI} \rangle \text{Doc}$$

### Example-4:

Let G be a grammar with the set of variables:

$$V = \{S, \langle \text{Noun phrase} \rangle, \langle \text{Verb phrase} \rangle, \langle \text{Adjective phrase} \rangle, \langle \text{Noun} \rangle, \langle \text{Verb} \rangle, \langle \text{Adjective} \rangle\}$$

The alphabet set:

$$\Sigma = \{\text{big, stout, John, bought, white, car, Jim, cheese, ate, green}\} \text{ with the rules:}$$

$$(1) S \rightarrow \langle \text{Noun phrase} \rangle \langle \text{Verb phrase} \rangle$$

$$(2) \langle \text{Noun phrase} \rangle \rightarrow \langle \text{Noun} \rangle | \langle \text{Adjective phrase} \rangle \langle \text{Noun} \rangle | \lambda$$

$$(3) \langle \text{Verb phrase} \rangle \rightarrow \langle \text{Verb} \rangle \langle \text{Noun phrase} \rangle$$

$$(4) \langle \text{Adjective phrase} \rangle \rightarrow \langle \text{Adjective phrase} \rangle \langle \text{Adjective} \rangle | \lambda$$

$$(5) \langle \text{Noun} \rangle \rightarrow \text{John} | \text{car} | \text{Jim} | \text{cheese}$$

$$(6) \langle \text{Verb} \rangle \rightarrow \text{bought} | \text{ate}$$

$$(7) \langle \text{Adjective} \rangle \rightarrow \text{big} | \text{stout} | \text{white} | \text{green}$$

Then the grammar generates, in particular, the following strings:

John bought car

Jim ate cheese big

Jim ate green cheese

John bought big car big stout John bought big white car

Unfortunately, the grammar also generates sentences like:

Big stout car bought big stout car

Big cheese ate Jim

Green Jim ate green big Jim

## 8.5 The Pumping Lemma for Context-Free Languages

In the theory of automata, if we want to prove that something is not a context-free language, it requires either finding a context-free grammar to describe the language or using another proof technique. For the second purpose the pumping lemma is the most commonly used concept. There is a common lemma to prove that a language is not context-free is the **Pumping Lemma for Context-Free Languages**.

**Theorem:**

**In the context-free languages the pumping lemma** states that if there is a language  $L$  is context-free, there exists some integer pumping length  $p \geq 1$  such that every string  $s \in L$  has a length of  $p$  or more symbols,  $|s| \geq p$ , that can be written  $s = uvwxy$  where  $u, v, w, x$ , and  $y$  are substrings of  $s$  such that:

- $|vwx| \leq p$
- $|vx| \geq 1$
- $uv^nwx^ny \in L \forall n \geq 0$

In the field of theory of automata, we know that all context-free languages are “pumpable”. It means that the pumping lemma constraints hold true for all context-free languages. In the case of a language is not pumpable, then it is not a context-free language. However, if a language is pumpable, it is not necessarily a context-free language. Because the set of regular languages is contained in the set of context-free languages, all regular languages must be pumpable too.

Basically, the pumping lemma grasps that arbitrarily long strings  $s \in L$  can be pumped without ever generating a new string that is not in the language  $L$ .

For the purpose of proving that a language is not context-free, there are two ways one is proof by contradiction and second is the pumping lemma. Set up a proof that claims that  $L$  is context-free, and show that a contradiction of the pumping lemma’s constraints arises in at least one of the three constraints listed above.

Essentially, the idea behind the pumping lemma for context-free languages is that there are certain constraints a language must adhere to in order to be a context-free language. For testing purpose, you can use the pumping lemma if all of these constraints hold for a particular language, and if they do not, you can prove with contradiction that the language is not context-free.



---

## 8.6 Summary

---

In this unit you have learnt about Closure Properties of Context Free Language, Decision Properties of Context Free Language, Application of Context Free Grammar, and Pumping Lemma for Context Free Language.

- A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple  $(N, T, P, S)$  where  $N$  is a set of non-terminal symbols.  $T$  is a set of terminals where  $N \cap T = \text{NULL}$ .  $P$  is a set of rules,  $P: N \rightarrow (N \cup T)^*$ , i.e., the left-hand side of the production rule  $P$  does not have any right context or left context.  $S$  is the start symbol.
- Context-free languages are closed under – Union, Concatenation, Kleene Star operation.
- Decision Properties of CFG are Test for Membership: Decidable. Test for Emptiness: Decidable, Test for finiteness: Decidable.
- Pumping Lemma for CFL states that for any Context Free Language  $L$ , it is possible to find two substrings that can be ‘pumped’ any number of times and still be in the same language. For any language  $L$ , we break its strings into five parts and pump second and fourth substring.
- Pumping Lemma, here also, is used as a tool to prove that a language is not CFL. Because, if any one string does not satisfy its conditions, then the language is not CFL.

---

## 8.7 Review Questions

---

- Q1. What are the closure properties of CFL? Define with suitable example.
- Q2. Prove that the language  $L = \{0^i 1 2^i \mid i \geq 0\}$  over the alphabet  $\{0, 1, 2\}$  is recursive and deterministic CFL
- Q3. What are the decision properties of CFL? Explain.
- Q4. Write the regular expression for the language over  $\Sigma = \{0\}$  having even length of the string.
- Q5. Write the regular expression for the language containing the string over  $\{0, 1\}$  in which there are at least two occurrences of 1's between any two occurrences of 1's between any two occurrences of 0's.





Uttar Pradesh Rajarshi Tandon  
Open University

# MCS - 113

## Master of Computer Science

### Theory of Computation

# Block

# 4

## Pushdown Automata and Turing Machine

---

### Unit - 9

Push down Automata	136
--------------------	-----

---

### Unit - 10

Turing Machine	149
----------------	-----

---

### Unit - 11

Undecidability	175
----------------	-----

---

---

## Course Design Committee

---

<b>Prof. Ashutosh Gupta</b> Director (In-charge) School of Computer and Information Sciences, UPRTOU Prayagraj	<b>Chairman</b>
<b>Prof. R. S. Yadav</b> Department of Computer Science and Engineering MNNIT Prayagraj	<b>Member</b>
<b>Dr. Marisha</b> Assistant Professor (Computer Science), School of Sciences, UPRTOU Prayagraj	<b>Member</b>
<b>Mr. Manoj Kumar Balwant</b> Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	<b>Member</b>

---

## Course Preparation Committee

---

<b>Dr. Ravi Shankar Shukla</b> Associate Professor Department of CSE, Invertis University Bareilly-243006, Uttar Pradesh	<b>Author</b>
<b>Prof. Abhay Saxena</b> Professor and Head, Department of Computer Science Dev Sanskriti Vishwavidyalaya, Hardwar, Uttarakhand	<b>Editor</b>
<b>Prof. Ashutosh Gupta</b> Director (In-charge) School of Computer and information, Sciences, UPRTOU Prayagraj	
<b>Mr. Manoj Kumar Balwant</b> Assistant Professor (computer science), School of Sciences, UPRTOU Prayagraj	<b>Course Coordinator</b>

---

© UPRTOU , Prayagraj - 2023

© MCS - 113 Theory of Computation

ISBN :

---

All Rights are reserved. No Part of this work may reproduced in any form, by mimeograph or any other means, without permission in writing from the Uttar Pradesh Rajarshi Tandon Open University.

Printed and Published by Vinay Kumar, Registrar, Uttar Pradesh rajarshi Tandon Open University, Prayagraj - 2023.

**Printed By. – M/s K.C.Printing & Allied Works, Panchwati, Mathura -281003.**

---

## Block-Introduction

---

This is the fourth block on Theory of computation and having detail description of Pushdown Automata and Turing Machine. Pushdown Automata is a finite automaton with extra memory called stack which helps Pushdown automata to recognize Context Free Languages. A Pushdown Automata (PDA) can be defined as: In a given state, PDA will read input symbol and stack symbol (top of the stack) and move to a new state and change the symbol of stack.

If we talk about Turing Machine, A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. ... After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left.

So we will begin the first unit with Pushdown automata. In this Unit, firstly we will discuss about formal definition of pushdown automata. We will also discuss about automata accepted by final state and empty state and finally we will cover equivalence between CFG and PDA.

Second unit begins with Turing Machine. In this Unit, you will know all about Turing machine and its formal definitions, and the behavior of Turing machine. You will also learn about how to draw transition diagram. After that, you will know about instantaneous description and language of a Turing Machine. In this unit you will learn about the variants of Turing machine and Universal Turing Machine. This unit covers about the Halting Problem and Church Thesis.

In the third unit, we provide another important topic i.e. Undecidability. In this unit we will provide the complete knowledge of recursive enumerable and undecidable problem about Turing Machines. We will also provide the complete information about unsolvable problems in Turing machines.

As you study the material, you will find that figures, tables are properly used and these will help to understand the concept. There are many sections in the units to easily understand the topic. Every unit has summary and review questions in the end of the unit which will help you to review yourself.

In your study, you will find that every unit has different equal length and your study time will vary for each unit.

We hope you enjoy studying the material and once again wish you all the best for your success.

---

## UNIT-9 Push Down Automata

---

### Structure

- 9.0 Introduction
- 9.1 Formal Definition of Pushdown Automata
- 9.2 Pushdown Automata accepted by final state and empty state
- 9.3 Equivalence between CFG and PDA.
- 9.4 Summary
- 9.5 Review Questions

### 9.0 Introduction

This is the first unit of this block. This unit explain the concept of push down automata. There are five sections in this unit. In the section 1.1, you will know about Formal Definition of Pushdown Automata. In the Section 1.2, you will learn about Pushdown Automata accepted by final state and empty state. Section 1.3 provide the detail knowledge about the Equivalence between CFG and PDA. Summary and review questions has been provided in the section 1.4 and 1.5 respectively.

### Objective

After studying this unit, you should be able to know about:

- Formal Definition of Pushdown Automata
- Pushdown Automata accepted by final state and empty state
- Equivalence between CFG and PDA.

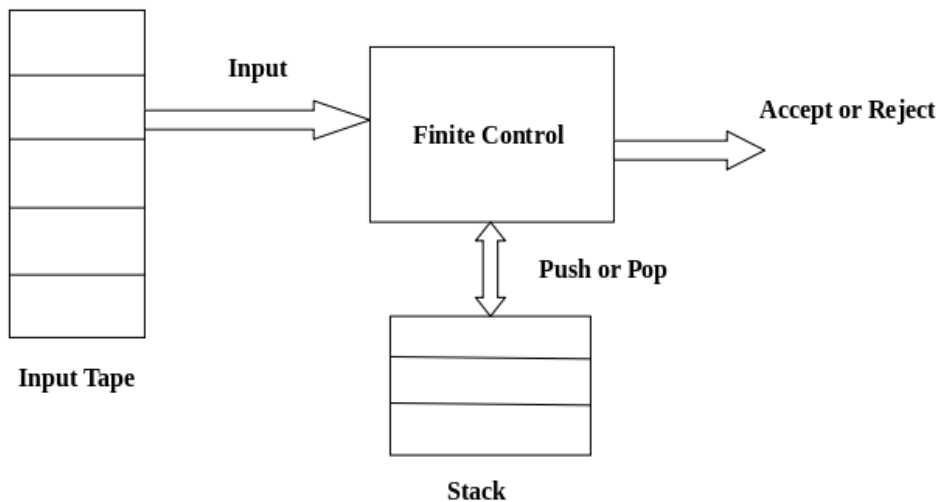
### 9.1 Pushdown Automata:

Pushdown automata is a way to implement a CFG in the same way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Pushdown automata is a part of an NFA which increased with an "external stack memory". With the help of stack, you can get a last-in-first-out memory management capability to Pushdown automata. Pushdown automata can store

an unbounded amount of information on the stack. It can access a limited amount of information on the stack. A PDA can push an element onto the top of the stack and pop off an element from the top of the stack. To read an element into the stack, the top elements must be popped off and are lost.

If we compare PDA and FA than we will find that a PDA is more powerful than FA. Any language which is accepted by FA can also be accepted by PDA but one important point is also noted here that a PDA also accepts a class of language which even cannot be accepted by FA. Thus PDA is much more powerful and superior to FA.



**Figure 9.1: Pushdown Automata**

### **PDA Components:**

**Input tape:** The input tape is divided in many cells or symbols. The input head is read-only and may only move from left to right, one symbol at a time.

**Finite control:** The finite control has some pointer which, points the current symbol which is to be read.

**Stack:** The stack is a structure in which we can push and remove the items from one end only. It has an infinite size. In PDA, the stack is used to store the items temporarily.

### **Formal Definition:**

The PDA can be defined as a collection of 7 components:

Q: the finite set of states

$\Sigma$ : the input set

$\Gamma$ : a stack symbol which can be pushed and popped from the stack

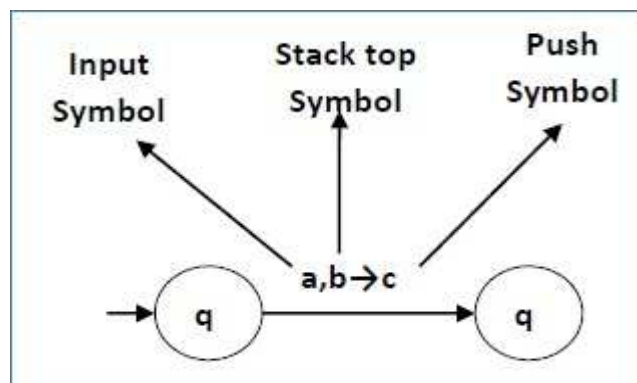
$q_0$ : the initial state

$Z$ : a start symbol which is in  $\Gamma$ .

$F$ : a set of final states

$\delta$ : mapping function which is used for moving from current state to next state.

The following diagram shows a transition in a PDA from a state  $q_1$  to state  $q_2$ , labelled as  $a, b \rightarrow c$  –



**Figure 2: Transition in PDA**

This means at state  $q_1$ , if we encounter an input string ‘a’ and top symbol of the stack is ‘b’, then we pop ‘b’, push ‘c’ on top of the stack and move to state  $q_2$ .

### **Instantaneous Description (ID)**

ID is an informal notation of how a PDA computes an input string and make a decision that string is accepted or rejected.

An instantaneous description is a triple  $(q, w, \alpha)$  where:

$q$  describes the current state.

$w$  describes the remaining input.

$\alpha$  describes the stack contents, top at the left.

### **Turnstile Notation:**

$\vdash$  sign describes the turnstile notation and represents one move.

$\vdash^*$  sign describes a sequence of moves.



For example,

$$(p, b, T) \vdash (q, w, \alpha)$$

In the above example, while taking a transition from state  $p$  to  $q$ , the input symbol 'b' is consumed, and the top of the stack 'T' is represented by a new string  $\alpha$ .

### Example 1:

**Design a PDA for accepting a language  $\{a^n b^{2n} \mid n \geq 1\}$ .**

Solution: In this language,  $n$  number of a's should be followed by  $2n$  number of b's. Hence, we will apply a very simple logic, and that is if we read single 'a', we will push two a's onto the stack. As soon as we read 'b' then for every single 'b' only one 'a' should get popped from the stack.

The ID can be constructed as follows:

$$\delta(q_0, a, Z) = (q_0, aaZ)$$

$$\delta(q_0, a, a) = (q_0, aaa)$$

Now when we read b, we will change the state from  $q_0$  to  $q_1$  and start popping corresponding 'a'. Hence,

$$\delta(q_0, b, a) = (q_1, \epsilon)$$

Thus this process of popping 'b' will be repeated unless all the symbols are read. Note that popping action occurs in state  $q_1$  only.

$$\delta(q_1, b, a) = (q_1, \epsilon)$$

After reading all b's, all the corresponding a's should get popped. Hence when we read  $\epsilon$  as input symbol then there should be nothing in the stack. Hence the move will be:

$$\delta(q_1, \epsilon, Z) = (q_2, \epsilon)$$

Where

$$PDA = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, Z\}, \delta, q_0, Z, \{q_2\})$$

We can summarize the ID as:

$$\delta(q_0, a, Z) = (q_0, aaZ)$$

$$\delta(q_0, a, a) = (q_0, aaa)$$

$$\delta(q_0, b, a) = (q_1, \epsilon)$$

$$\delta(q_1, b, a) = (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, Z) = (q_2, \epsilon)$$

Now we will simulate this PDA for the input string "aaabbbbbb".

$\delta(q_0, aaabbbbbb, Z) \vdash \delta(q_0, aabbbbbb, aaZ)$   
 $\vdash \delta(q_0, abbbbbb, aaaaZ)$   
 $\vdash \delta(q_0, bbbbbb, aaaaaaZ)$   
 $\vdash \delta(q_1, bbbbbb, aaaaaaZ)$   
 $\vdash \delta(q_1, bbbb, aaaaZ)$   
 $\vdash \delta(q_1, bbb, aaaZ)$   
 $\vdash \delta(q_1, bb, aaZ)$   
 $\vdash \delta(q_1, b, aZ)$   
 $\vdash \delta(q_1, \epsilon, Z)$   
 $\vdash \delta(q_2, \epsilon)$   
 ACCEPT

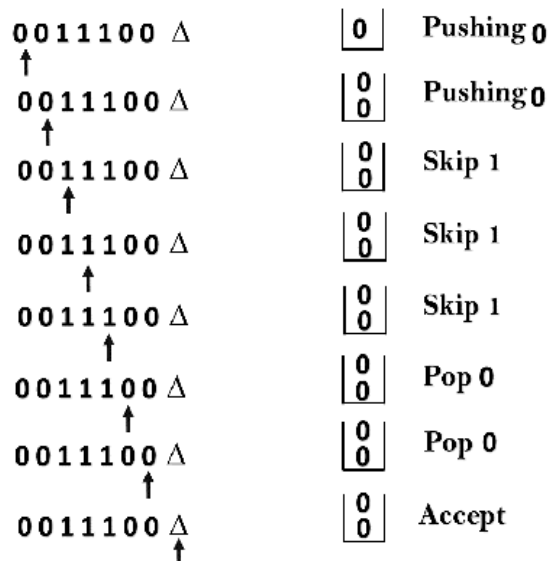
### Example 2:

**Design a PDA for accepting a language  $\{0^n 1^m 0^n \mid m, n \geq 1\}$ .**

**Solution:** In this PDA,  $n$  number of 0's are followed by any number of 1's followed  $n$  number of 0's. Hence the logic for design of such PDA will be as follows:

Push all 0's onto the stack on encountering first 0's. Then if we read 1, just do nothing. Then read 0, and on each read of 0, pop one 0 from the stack.

For instance:



This scenario can be written in the ID form as:

$$\delta(q_0, 0, Z) = \delta(q_0, 0Z)$$

$$\delta(q_0, 0, 0) = \delta(q_0, 00)$$

$$\delta(q_0, 1, 0) = \delta(q_1, 0)$$

$$\delta(q_0, 1, 0) = \delta(q_1, 0)$$

$$\delta(q_1, 0, 0) = \delta(q_1, \varepsilon)$$

$$\delta(q_0, \varepsilon, Z) = \delta(q_2, Z) \quad (\text{ACCEPT state})$$

Now we will simulate this PDA for the input string "0011100".

$$\delta(q_0, 0011100, Z) \vdash \delta(q_0, 011100, 0Z)$$

$$\vdash \delta(q_0, 11100, 00Z)$$

$$\vdash \delta(q_0, 1100, 00Z)$$

$$\vdash \delta(q_1, 100, 00Z)$$

$$\vdash \delta(q_1, 00, 00Z)$$

$$\vdash \delta(q_1, 0, 0Z)$$

$$\vdash \delta(q_1, \varepsilon, Z)$$

$$\vdash \delta(q_2, Z)$$

ACCEPT

---

## Check your progress

---

Q1. What do you understand by PDA? Explain with suitable example.

Q2. How do you write PDA?

## 9.2 PDA Acceptance

A language can be accepted by Pushdown automata using two approaches:

1. Final State acceptance: The PDA is supposed to accept its input by the final state if it enters any final state in zero or more moves after reading the entire input.

Let  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$  be a PDA. The language acceptable by the final state can be defined as:

$$L(PDA) = \{w \mid (q_0, w, Z) \vdash^* (p, \varepsilon, \varepsilon), q \in F\}$$

2. Acceptance by Empty Stack: On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty.

Let  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$  be a PDA. The language acceptable by empty stack can be defined as:

$$N(PDA) = \{w \mid (q_0, w, Z) \vdash^* (p, \varepsilon, \varepsilon), q \in Q\}$$

Equivalence of Acceptance by Final State and Empty Stack

- If  $L = N(P_1)$  for some PDA  $P_1$ , then there is a PDA  $P_2$  such that  $L = L(P_2)$ . That means the language accepted by empty stack PDA will also be accepted by final state PDA.
- If there is a language  $L = L(P_1)$  for some PDA  $P_1$  then there is a PDA  $P_2$  such that  $L = N(P_2)$ . That means language accepted by final state PDA is also acceptable by empty stack PDA.

**Example:**

**Construct a PDA that accepts the language  $L$  over  $\{0, 1\}$  by empty stack which accepts all the string of 0's and 1's in which a number of 0's are twice of number of 1's.**

**Solution:**

There are two parts for designing this PDA:

- If 1 comes before any 0's
- If 0 comes before any 1's.

We are going to design the first part i.e. 1 comes before 0's. The logic is that read single 1 and push two 1's onto the stack. Thereafter on reading two 0's, POP two 1's from the stack. The  $\delta$  can be

$$\delta(q_0, 1, Z) = (q_0, 11, Z) \quad \text{Here } Z \text{ represents that stack is empty}$$

$$\delta(q_0, 0, 1) = (q_0, \varepsilon)$$

Now, consider the second part i.e. if 0 comes before 1's. The logic is that read first 0, push it onto the stack and change state from  $q_0$  to  $q_1$ . [Note that state  $q_1$  indicates that first 0 is read and still second 0 has yet to read].

Being in  $q_1$ , if 1 is encountered then POP 0. Being in  $q_1$ , if 0 is read then simply read that second 0 and move ahead. The  $\delta$  will be:

$$\delta(q_0, 0, Z) = (q_1, 0Z)$$

$$\delta(q_1, 0, 0) = (q_1, 0)$$

$$\delta(q_1, 0, Z) = (q_0, \varepsilon)$$

*(indicate that one 0 and one 1 is already read, so simply read the second 0)*

$$\delta(q_1, 1, 0) = (q_1, \varepsilon)$$

Now, summarize the complete PDA for given L is:

$$\delta(q_0, 1, Z) = (q_0, 11Z)$$

$$\delta(q_0, 0, 1) = (q_1, \varepsilon)$$

$$\delta(q_0, 0, Z) = (q_1, 0Z)$$

$$\delta(q_1, 0, 0) = (q_1, 0)$$

$$\delta(q_1, 0, Z) = (q_0, \varepsilon)$$

$$\delta(q_0, \varepsilon, Z) = (q_0, \varepsilon) \quad \text{ACCEPT state}$$

### **Non-deterministic Pushdown Automata**

The non-deterministic pushdown automata is very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG accepts both deterministic PDA and non-deterministic PDAs as well. In the same manner, there are some CFGs which can be accepted only by NPDA and not by DPDA. Thus NPDA is more commanding than DPDA.

#### **Example:**

#### **Design PDA for Palindrome strips.**

#### **Solution:**

Suppose the language consists of string

$$L = \{aba, aa, bb, bab, bbabb, aabaa, \dots\}.$$

The string can be odd palindrome or even palindrome. The logic for constructing PDA is that we will push a symbol onto the stack till half of the string then we will read each symbol and then perform the pop operation. We will compare to see whether the symbol which is popped is similar to the symbol which is read whether we reach to end of the input, we expect the stack to be empty.

This type of PDA is a non-deterministic PDA because it finds the mid from the given string and start reading from left and matching it with from right (reverse) direction leads to non-deterministic moves. Here is the ID.

1. $\delta(q_1, a, Z) = (q_1, aZ)$	Pushing the symbols onto the stack
2. $\delta(q_1, b, Z) = (q_1, bZ)$	
3. $\delta(q_1, a, a) = (q_1, aa)$	
4. $\delta(q_1, a, b) = (q_1, ab)$	
5. $\delta(q_1, a, b) = (q_1, ba)$	
6. $\delta(q_1, b, b) = (q_1, bb)$	
7. $\delta(q_1, a, a) = (q_2, \epsilon)$	Popping the symbols on reading the same kind of symbol
8. $\delta(q_1, b, b) = (q_2, \epsilon)$	
9. $\delta(q_2, a, a) = (q_2, \epsilon)$	
10. $\delta(q_2, b, b) = (q_2, \epsilon)$	
11. $\delta(q_2, \epsilon, Z) = (q_2, \epsilon)$	

### Simulation of abaaba

$\delta(q_1, abaaba, Z)$	Apply rule 1
$\vdash \delta(q_1, baaba, aZ)$	Apply rule 5
$\vdash \delta(q_1, aaba, baZ)$	Apply rule 4
$\vdash \delta(q_1, aba, abaZ)$	Apply rule 7
$\vdash \delta(q_2, ba, baZ)$	Apply rule 8
$\vdash \delta(q_2, a, aZ)$	Apply rule 7
$\vdash \delta(q_2, \epsilon, Z)$	Apply rule 11
$\vdash \delta(q_2, \epsilon)$	Accept

## 9.3 Equivalence between CFG and PDA.

The first symbol on R.H.S. production must be a terminal symbol. The following steps are used to obtain PDA from CFG is:

**Step 1:** Convert the given productions of CFG into GNF.

**Step 2:** The PDA will only have one state  $\{q\}$ .

**Step 3:** The initial symbol of CFG will be the initial symbol in the PDA.

**Step 4:** For non-terminal symbol, add the following rule:

$$\delta(q, \epsilon, A) = (q, \alpha)$$

Where the production rule is  $A \rightarrow \alpha$

**Step 5:** For each terminal symbols, add the following rule:

$$\delta(q, a, a) = (q, \epsilon) \text{ for every terminal symbol}$$

**Example 1:**

**Convert the following grammar to a PDA that accepts the same language.**

$$S \rightarrow 0S1 \mid A$$

$$A \rightarrow 1A0 \mid S \mid \epsilon$$

**Solution:**

The CFG can be first simplified by eliminating unit productions:

$$S \rightarrow 0S1 \mid 1S0 \mid \epsilon$$

Now we will convert this CFG to GNF:

$$S \rightarrow 0SX \mid 1SY \mid \epsilon$$

$$X \rightarrow 1$$

$$Y \rightarrow 0$$

The PDA can be:

$$R1: \delta(q, \epsilon, S) = \{(q, 0SX) \mid (q, 1SY) \mid (q, \epsilon)\}$$

$$R2: \delta(q, \epsilon, X) = \{(q, 1)\}$$

$$R3: \delta(q, \epsilon, Y) = \{(q, 0)\}$$

$$R4: \delta(q, 0, 0) = \{(q, \epsilon)\}$$

$$R5: \delta(q, 1, 1) = \{(q, \epsilon)\}$$

**Example 2:**

**Construct PDA for the given CFG, and test whether 0104 is acceptable by this PDA.**

$$S \rightarrow 0BB$$

$$B \rightarrow 0S \mid 1S \mid 0$$

**Solution:**

The PDA can be given as:

$$A = \{(q), (0, 1), (S, B, 0, 1), \delta, q, S?\}$$

The production rule  $\delta$  can be:

$$R1: \delta (q, \varepsilon, S) = \{(q, 0BB)\}$$

$$R2: \delta (q, \varepsilon, B) = \{(q, 0S) \mid (q, 1S) \mid (q, 0)\}$$

$$R3: \delta (q, 0, 0) = \{(q, \varepsilon)\}$$

$$R4: \delta (q, 1, 1) = \{(q, \varepsilon)\}$$

Testing 0104 i.e. 010000 against PDA:

$$\delta (q, 010000, S) \vdash \delta (q, 010000, 0BB)$$

$$\vdash \delta (q, 10000, BB) \quad R1$$

$$\vdash \delta (q, 10000, 1SB) \quad R3$$

$$\vdash \delta (q, 0000, SB) \quad R2$$

$$\vdash \delta (q, 0000, 0BBB) \quad R1$$

$$\vdash \delta (q, 000, BBB) \quad R3$$

$$\vdash \delta (q, 000, 0BB) \quad R2$$

$$\vdash \delta (q, 00, BB) \quad R3$$

$$\vdash \delta (q, 00, 0B) \quad R2$$

$$\vdash \delta (q, 0, B) \quad R3$$

$$\vdash \delta (q, 0, 0) \quad R2$$

$$\vdash \delta (q, \varepsilon) \quad R3$$

ACCEPT



**Example 3:****Draw a PDA for the CFG given below:**

$$S \rightarrow aSb$$

$$S \rightarrow a \mid b \mid \varepsilon$$

**Solution:**

The PDA can be given as:

$$P = \{(q), (a, b), (S, a, b, z_0), \delta, q, z_0, q\}$$

**The mapping function  $\delta$  will be:**

$$R1: \delta(q, \varepsilon, S) = \{(q, aSb)\}$$

$$R2: \delta(q, \varepsilon, S) = \{(q, a) \mid (q, b) \mid (q, \varepsilon)\}$$

$$R3: \delta(q, a, a) = \{(q, \varepsilon)\}$$

$$R4: \delta(q, b, b) = \{(q, \varepsilon)\}$$

$$R5: \delta(q, \varepsilon, z_0) = \{(q, \varepsilon)\}$$

**Simulation: Consider the string aaabb**

$$\delta(q, \varepsilon aaabb, S) \vdash \delta(q, aaabb, aSb) \quad R3$$

$$\vdash \delta(q, \varepsilon aabb, Sb) \quad R1$$

$$\vdash \delta(q, aabb, aSbb) \quad R3$$

$$\vdash \delta(q, \varepsilon abb, Sbb) \quad R2$$

$$\vdash \delta(q, abb, abb) \quad R3$$

$$\vdash \delta(q, bb, bb) \quad R4$$

$$\vdash \delta(q, b, b) \quad R4$$

$$\vdash \delta(q, \varepsilon, z_0) \quad R5$$

$$\vdash \delta(q, \varepsilon) \quad \text{ACCEPT}$$

---

## 9.4 Summary

---

In this unit you have learnt about pushdown automata and formal definition about pushdown automata. You have also learnt about Pushdown Automata accepted by final state and empty state, and finally you have learnt about Equivalence between CFG and PDA.

- In the theory of computation, a branch of theoretical computer science, a pushdown automaton (PDA) is a type of automaton that employs a stack.
- Pushdown automata are used in theories about what can be computed by machines. They are more capable than finite-state machines but less capable than Turing machines.
- Deterministic pushdown automata can recognize all deterministic context-free languages while nondeterministic ones can recognize all context-free languages, with the former often used in parser design.
- Every context-free grammar can be transformed into an equivalent nondeterministic pushdown automaton.
- The derivation process of the grammar is simulated in a leftmost way. Where the grammar rewrites a nonterminal, the PDA takes the topmost nonterminal from its stack and replaces it by the right-hand part of a grammatical rule (expand). Where the grammar generates a terminal symbol, the PDA reads a symbol from input when it is the topmost symbol on the stack (match). In a sense the stack of the PDA contains the unprocessed data of the grammar, corresponding to a pre-order traversal of a derivation tree.

---

## 9.5 Review Questions

---

Q1. Give pushdown automata that recognize the following languages. Give both a drawing and 6-tuple specification for each PDA.

$$A = \{w \in \{0, 1\}^* \mid w \text{ contains at least three } 1\text{'s}\}$$

Q2. Define the pushdown automata for language  $\{a^n b^n \mid n > 0\}$

Q3. Construct a PDA for language  $L = \{0^n 1^m 2^m 3^n \mid n \geq 1, m \geq 1\}$

Q4. Construct a PDA for language  $L = \{0^n 1^m \mid n \geq 1, m \geq 1, m > n+2\}$

Q5. Draw a PDA for the CFG given below:

$$S \rightarrow aSb$$

$$S \rightarrow a \mid b \mid \varepsilon$$

---

# UNIT-10 Turing Machine

---

## Structure

10.0 Introduction

10.1 Turing Machine (TM) –Formal Definition

10.2 Behaviour of Turing Machine

10.3 Transition diagram of Turing Machine

10.4 Instantaneous Description

10.5 Language of a TM

10.6 Variants of TM

10.7 Universal Turing Machine

10.7.1 Interchangeability of program and behaviour: a notation

10.7.2 Interchangeability of program and behaviour: a basic set of functions

10.8 Halting Problem

10.9 Church Thesis.

10.10 Summary

10.11 Review Questions

## 10.0 Introduction

This unit is the second unit of this block. This unit provide complete knowledge of Turing Machine. There are eleven sections in this unit. Formal definition has been defined in the section 2.1. Next section 2.2 describe behaviour of Turing machine. In the section 2.3, you will learn about transition diagram of Turing machine. Section 2.4 explain Instantaneous Description of Turing machine. In the section 2.5, you will get the Language of a Turing Machine. Variants of Turing Machine explain in the section 2.6, Next section i.e. Section 2.7 Universal Turing Machine also have two sub sections. These sub sections provide a notation of Interchangeably of program and behaviour and basic set of functions of Interchangeably of program and behaviour. Section 2.9 shows Halting Problem and Section 2.9 provide Church Thesis. Last two sections i.e. Section 2.10 and 2.11 provide summary and reviews questions.

## Objective

After studying this unit, you should be able to define:

- Formal Definition of Turing Machine and behaviour of Turing Machine.
- Transition diagram of Turing Machine.

- Instantaneous Description and Language of a Turing Machine.
- Variants of Turing Machine and Universal Turing Machine.
- Halting Problem and Church Thesis.

## 10.1 Turing Machine (TM):

A Turing machine is a mathematical model of computation that defines an abstract machine, which manipulates symbols on a strip of tape according to a table of rules. Despite the model's simplicity, given by any computer algorithm, a Turing machine is capable of simulating that algorithm's logic can be construct.

The machine operates on an infinite memory tape divided into discrete "cells". The machine positions its "head" over a cell and "reads" or "scans" the symbol there. Then, as per the symbol and the machine's own present state in a "finite table" of user-specified instructions, the machine

- (i) Writes a symbol (e.g., a digit or a letter from a finite alphabet) in the cell (some models allow symbol erasure or no writing)
- (ii) Either moves the tape one cell left or right (some models allow no motion, some models move the head)
- (iii) Either proceeds to a subsequent instruction or halts the computation.

The Turing machine was invented in 1936 by Alan Turing, who called it an "a-machine" (automatic machine). With this model, Turing was able to answer two questions:

- (1) Does a machine exist that can determine whether any arbitrary machine on its tape is "circular" (e.g., freezes, or fails to continue its computational task)?
- (2) Does a machine exist that can determine whether any arbitrary machine on its tape ever prints a given symbol?

Thus by providing a mathematical description of a very simple device capable of arbitrary computations, he was able to prove properties of computation in general—and in particular, the uncomputability of the Entscheidungs problem ('decision problem').

With the help of Turing machines, it has been proved the existence of fundamental limitations on the power of mechanical computation. Although they can express arbitrary computations, their minimalist design makes them unsuitable for computation in practice: real-world computers are based on different designs that, unlike Turing machines, use random-access memory.

For any System Turing completeness is the capability of instruction to simulate a Turing machine. A programming language that is Turing complete is theoretically capable of stating all tasks accomplishable by computers; if the limitations of finite memories are ignored then all programming languages will be Turing complete.

A Turing machine can be used as a general example of a central processing unit (CPU) that handles and manipulate all data, with the help of canonical machine to store data using sequential memory.

More specifically, it is an automaton machine which is capable of counting some arbitrary subset of valid strings of an alphabet; these arbitrary subset of strings is part of a recursively enumerable set. A Turing machine can perform read and write operations with the help of a tape of infinite length.

Let us assume a black box, the Turing machine cannot recognise whether it will eventually count any one specific string of the subset with a given program, because of the halting problem. As we know Halting problem is unsolvable, which has major implications for the theoretical limits of computing.

The Turing machine is also accomplished of handling an unrestricted grammar, which further implies that it is capable of robustly evaluating first-order logic in an infinite number of ways. This is demonstrated with the help of lambda calculus.

The term Universal Turing machine is able to simulate any other Turing machine. It is also known as UTM. This mathematically oriented definition with a similar "universal" nature was introduced by Alonzo Church. It works on lambda calculus intertwined with Turing's in a formal theory of computation known as the Church–Turing thesis. With the help of the thesis, we can understand that Turing machines truly capture the informal concept of effective methods in logic and mathematics. It also delivers an exact definition of an algorithm or "mechanical procedure". Studying their abstract properties yields many insights into computer science and complexity theory.

### **Formal definition of Turing machine**

A Turing machine can be defined as a collection of 7 components:

Q: the finite set of states

$\Sigma$ : the finite set of input symbols

T: the tape symbol

$q_0$ : the initial state

F: a set of final states

B: a blank symbol used as an end marker for input

$\delta$ : a transition or mapping function.

The mapping function shows the mapping from states of finite automata and input symbol on the tape to the next states, external symbols and the direction for moving the tape head. This is known as a triple or a program for Turing machine.

$$(q_0, a) \rightarrow (q_1, A, R)$$

That means in  $q_0$  state, if we read symbol 'a' then it will go to state  $q_1$ , replaced a by X and move ahead right (R stands for right).

### Example:

**Construct TM for the language  $L = \{0^n 1^n\}$  where  $n \geq 1$ .**

### Solution:

We have already solved this problem by PDA. In PDA, we have a stack to remember the previous symbol. The main advantage of the Turing machine is we have a tape head which can be moved forward or backward, and the input tape can be scanned.

The simple logic which we will apply is read out each '0' mark it by A and then move ahead along with the input tape and find out 1 convert it to B. Now, repeat this process for all a's and b's.

Now we will see how this Turing machine work for 0011.

The simulation for 0011 can be shown as below:

0	0	1	1	$\Delta$	-----
---	---	---	---	----------	-------

Now, we will see how this Turing machine will work for 0011. Initially, state is  $q_0$  and head points to 0 as:

0	0	1	1	$\Delta$
↑				

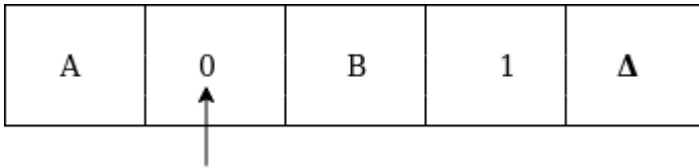
The move will be  $\delta(q_0, 0) = \delta(q_1, A, R)$  which means it will go to state  $q_1$ , replaced 0 by A and head will move to the right as:

A	0	1	1	$\Delta$
	↑			

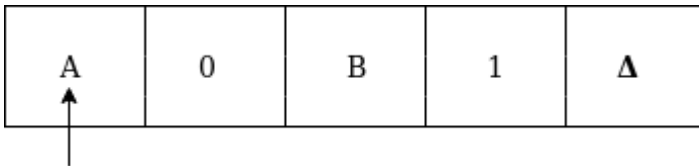
The move will be  $\delta(q_1, 0) = \delta(q_1, 0, R)$  which means it will not change any symbol, remain in the same state and move to the right as:

A	0	1	1	$\Delta$
		↑		

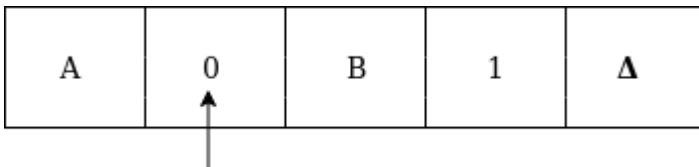
The move will be  $\delta(q_1, 1) = \delta(q_2, B, L)$  which means it will go to state  $q_2$ , replaced 1 by B and head will move to left as:



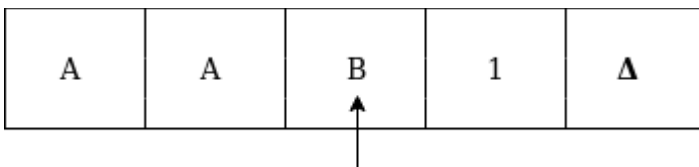
Now move will be  $\delta(q_2, 0) = \delta(q_2, 0, L)$  which means it will not change any symbol, remain in the same state and move to left as:



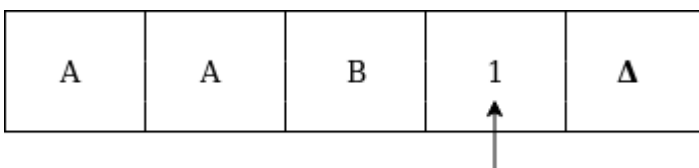
The move will be  $\delta(q_2, A) = \delta(q_0, A, R)$ , it means will go to state  $q_0$ , replaced A by A and head will move to the right as:



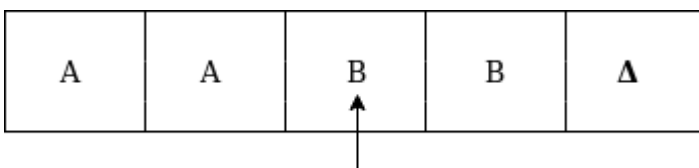
The move will be  $\delta(q_0, 0) = \delta(q_1, A, R)$  which means it will go to state  $q_1$ , replaced 0 by A, and head will move to right as:



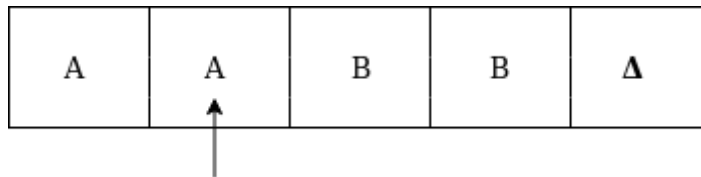
The move will be  $\delta(q_1, B) = \delta(q_1, B, R)$  which means it will not change any symbol, remain in the same state and move to right as:



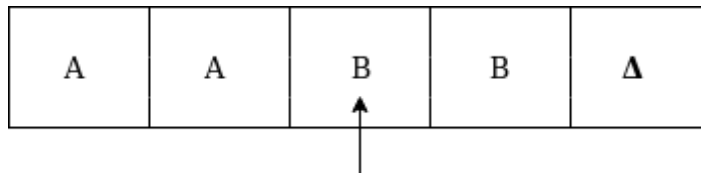
The move will be  $\delta(q_1, 1) = \delta(q_2, B, L)$  which means it will go to state  $q_2$ , replaced 1 by B and head will move to left as:



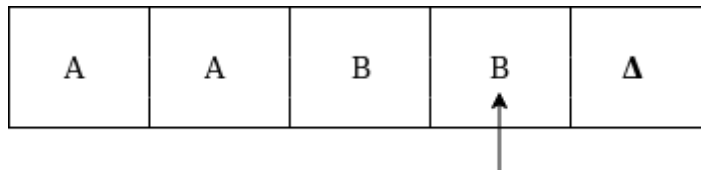
The move  $\delta(q_2, B) = (q_2, B, L)$  which means it will not change any symbol, remain in the same state and move to left as:



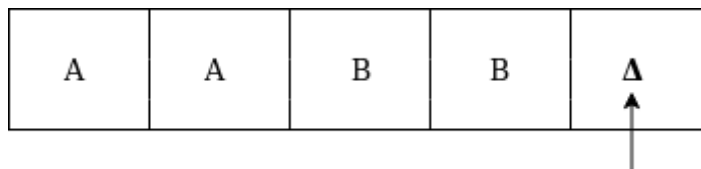
Now immediately before B is A that means all the 0's is marked by A. So we will move right to ensure that no 1 is present. The move will be  $\delta(q_2, A) = (q_0, A, R)$  which means it will go to state  $q_0$ , will not change any symbol, and move to right as:



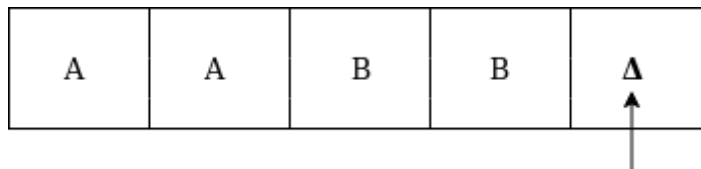
The move  $\delta(q_0, B) = (q_3, B, R)$  which means it will go to state  $q_3$ , will not change any symbol, and move to right as:



The move  $\delta(q_3, B) = (q_3, B, R)$  which means it will not change any symbol, remain in the same state and move to right as:



The move  $\delta(q_3, \Delta) = (q_4, \Delta, R)$  which means it will go to state  $q_4$  which is the HALT state and HALT state is always an accept state for any TM.



The same TM can be represented by Transition Diagram.

## 10.2 Behaviour of Turing Machine:

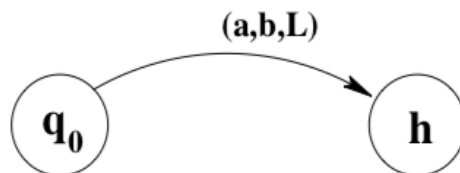
A Turing machine is deterministic or non-deterministic, it is depending upon the number of moves in transition. A transition is called deterministic (DTM) if a Turing Machine has at most one move in it. If there are one or more moves, then it is called non-deterministic TM (NTM or NDTM).



- A non-deterministic TM is equivalent to a deterministic TM.
- Some single tape TM simulates every two PDA (a PDA with 2 stacks).
- The read only TM may be considered as a Finite Automata (FA) with additional property of being able to move its head in both directions (left and right).

### 10.3 Transition Diagram of Turing Machine

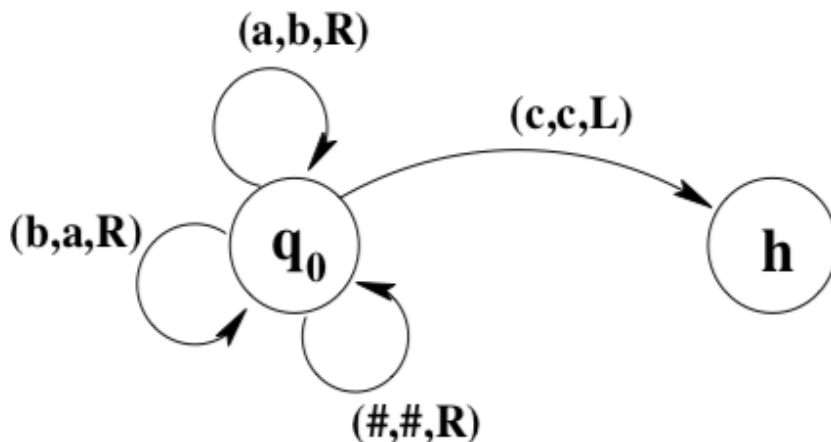
The transition diagram for a Turing machine is similar to the transition diagram for a DFA. However, there are no “accepting” states (only a halt state). Furthermore, there must be a way to specify the output symbol and the direction of motion for each step of the computation. We do this by labelling arrows with notations of the form  $(\sigma, \tau, L)$  and  $(\sigma, \tau, R)$ , where  $\sigma$  and  $\tau$  are symbols in the Turing machine’s alphabet. For example,



**Figure 10.1: Transition diagram of Turing Machine**

Indicates that when the machine is in state  $q_0$  and reads an  $a$ , it writes  $b$ , moves left, and enters state  $h$ .

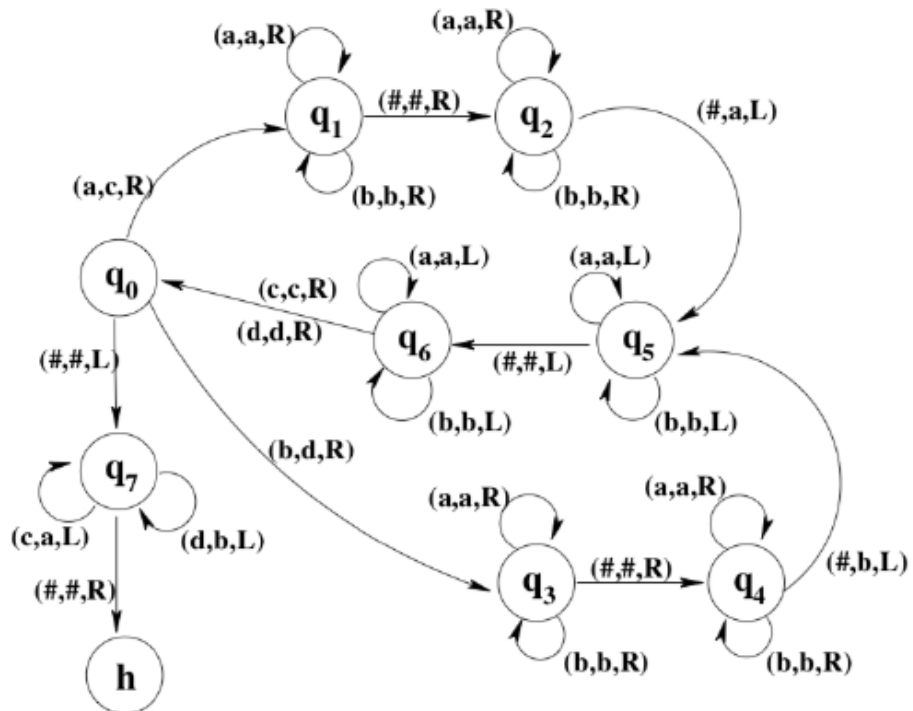
Here, for example, is a transition diagram for a simple Turing machine that moves to the right, changing  $a$ 's to  $b$ 's and vice versa, until it finds  $ac$ . It leaves blanks ( $\#$ 's) unchanged. When and if the machine encounters  $ac$ , it moves to the left and halts:



**Figure 10.2: Turing Machine transition diagram movement from right**

To simplify the diagrams, we will leave out any transitions that are not relevant to the computation that we want the machine to perform. You can assume that the action for any omitted transition is to write the same symbol that was read, move right, and halt.

For example, shown below is a transition diagram for a Turing machine that makes a copy of a string of a's and b's. To use this machine, you would write a string of a's and b's on its tape, place the machine on the first character of the string, and start the machine in its start state,  $q_0$ . When the machine halts, there will be two copies of the string on the tape, separated by a blank. The machine will be positioned on the first character of the leftmost copy of the string. Note that this machine uses c's and d's in addition to a's and b's. While it is copying the input string, it temporarily changes the a's and b's that it has copied to c's and d's, respectively. In this way it can keep track of which characters it has already copied. After the string has been copied, the machine changes the c's and d's back to a's and b's before halting.



**Figure 10.3: Transition diagram for a Turing machine that makes a copy of a string of a's and b's**

In this machine, state  $q_0$  checks whether the next character is an a, a b, or a # (indicating the end of the string). States  $q_1$  and  $q_2$  add an a to the end of the new string, and states  $q_3$  and  $q_4$  do the same thing with a b. States  $q_5$  and  $q_6$  return the machine to the next character in the input string. When the end of the input string is reached, state  $q_7$  will move the machine back to the start of the input string, changing c's and d's back to a's and b's as it goes. Finally, when the machine hits the # that precedes the input string, it moves to the right and halts. This leave it back at the first character of the input string. It would be a good idea to work through the execution of this machine for a few sample

input strings. You should also check that it works even for an input string of length zero.

Our primary interest in Turing machines is as language processors. Suppose that  $w$  is a string over an alphabet  $\Sigma$ . We will assume that  $\Sigma$  does not contain the blank symbol. We can use  $w$  as input to a Turing machine  $M = (Q, \Lambda, q_0, \delta)$  provided that  $\Sigma \subseteq \Lambda$ . To use  $w$  as input for  $M$  we will write  $w$  on  $M$ 's tape and assume that the remainder of the tape is blank. We place the machine on the cell containing the first character of the string, except that if  $w = \epsilon$  then we simply place the machine on a completely blank tape. Then we start the machine in its initial state,  $q_0$  and see what computation it performs. We refer to this setup as “running  $M$  with input  $w$ .”

When  $M$  is run with input  $w$ , it is possible that it will just keep running forever without halting. In that case, it doesn't make sense to ask about the output of the computation. Suppose however that  $M$  does halt on input  $w$ . Suppose, furthermore, that when  $M$  halts, its tape is blank except for a string  $x$  of non-blank symbols, and that the machine is located on the first character of  $x$ . In this case, we will say that “ $M$  halts with output  $x$ .” In addition, if  $M$  halts with an entirely blank tape, we say that “ $M$  halts with output  $\epsilon$ .” Note that when we run  $M$  with input  $w$ , one of three things can happen: (1)  $M$  might halt with some string as output; (1)  $M$  might fail to halt; or (3)  $M$  might halt in some configuration that doesn't count as outputting any string.

The fact that a Turing machine can produce an output value allows us for the first time to deal with computation of functions. A function  $f: A \rightarrow B$  takes an input value in the set  $A$  and produces an output value in the set  $B$ . If the sets are sets of strings, we can now ask whether the values of the function can be computed by a Turing Machine? That is, a Turing machine  $M$  such that, given any string  $w$  as an input,  $M$  will compute as its output the string  $f(w)$ . If this is the case, then we say that  $f$  is a Turing-computable function.

### Example 1:

**Construct a TM for the language  $L = \{0^n 1^n 2^n\}$  where  $n \geq 1$**

#### Solution:

$L = \{0^n 1^n 2^n \mid n \geq 1\}$  represents language where we use only 3 character, i.e., 0, 1 and 2. In this, some number of 0's followed by an equal number of 1's and then followed by an equal number of 2's. Any type of string which falls in this category will be accepted by this language.

The simulation for 001122 can be shown as below:

0	0	1	1	2	2	X	-----
---	---	---	---	---	---	---	-------

Now, we will see how this Turing machine will work for 001122. Initially, state is  $q_0$  and head points to 0 as:

0	0	1	1	2	2	X
---	---	---	---	---	---	---

↑

The move will be  $\delta(q_0, 0) = \delta(q_1, A, R)$  which means it will go to state  $q_1$ , replaced 0 by A and head will move to the right as:

A	0	1	1	2	2	X
---	---	---	---	---	---	---

↑

The move will be  $\delta(q_1, 0) = \delta(q_1, 0, R)$  which means it will not change any symbol, remain in the same state and move to the right as:

A	0	1	1	2	2	X
---	---	---	---	---	---	---

↑

The move will be  $\delta(q_1, 1) = \delta(q_2, B, R)$  which means it will go to state  $q_2$ , replaced 1 by B and head will move to right as:

A	0	B	1	2	2	X
---	---	---	---	---	---	---

↑

The move will be  $\delta(q_2, 1) = \delta(q_2, 1, R)$  which means it will not change any symbol, remain in the same state and move to right as:

A	0	B	1	2	2	X
---	---	---	---	---	---	---

↑

The move will be  $\delta(q_2, 2) = \delta(q_3, C, R)$  which means it will go to state  $q_3$ , replaced 2 by C and head will move to right as:

A	0	B	1	C	2	X
---	---	---	---	---	---	---

↑

Now move  $\delta(q_3, 2) = \delta(q_3, 2, L)$  and  $\delta(q_3, C) = \delta(q_3, C, L)$  and  $\delta(q_3, 1) = \delta(q_3, 1, L)$  and  $\delta(q_3, B) = \delta(q_3, B, L)$  and  $\delta(q_3, 0) = \delta(q_3, 0, L)$ , and then move  $\delta(q_3, A) = \delta(q_0, A, R)$ , it means it will go to state  $q_0$ , replaced A by A and head will move to right as:

A	0	B	1	C	2	X
---	---	---	---	---	---	---

↑

The move will be  $\delta(q_0, 0) = \delta(q_1, A, R)$  which means it will go to state  $q_1$ , replaced 0 by A, and head will move to right as:

A	A	B	1	C	2	X
---	---	---	---	---	---	---

↑

The move will be  $\delta(q_1, B) = \delta(q_1, B, R)$  which means it will not change any symbol, remain in the same state and move to right as:

A	A	B	1	C	2	X
---	---	---	---	---	---	---

↑

The move will be  $\delta(q_1, 1) = \delta(q_2, B, R)$  which means it will go to state  $q_2$ , replaced 1 by B and head will move to right as:

A	A	B	B	C	2	X
---	---	---	---	---	---	---

↑

The move will be  $\delta(q_2, C) = \delta(q_2, C, R)$  which means it will not change any symbol, remain in the same state and move to right as:

A	A	B	B	C	2	X
---	---	---	---	---	---	---

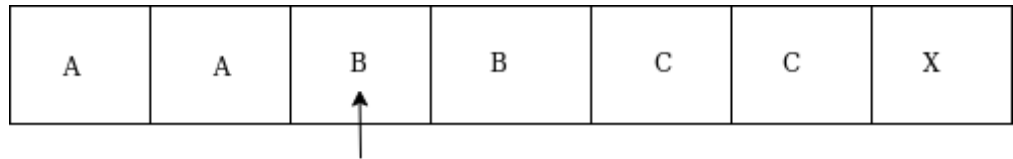
↑

The move will be  $\delta(q_2, 2) = \delta(q_3, C, L)$  which means it will go to state  $q_3$ , replaced 2 by C and head will move to left until we reached A as:

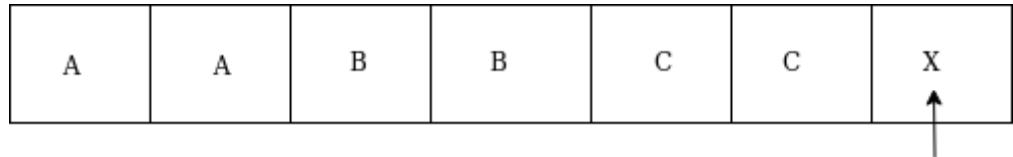
A	A	B	B	C	C	X
---	---	---	---	---	---	---

↑

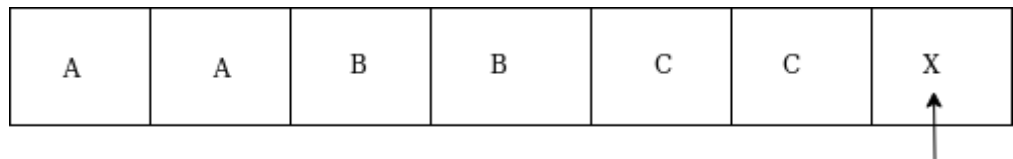
Immediately before B is A that means all the 0's are market by A. So we will move right to ensure that no 1 or 2 is present. The move will be  $\delta(q_3, B) = \delta(q_4, B, R)$  which means it will go to state  $q_4$ , will not change any symbol, and move to right as:



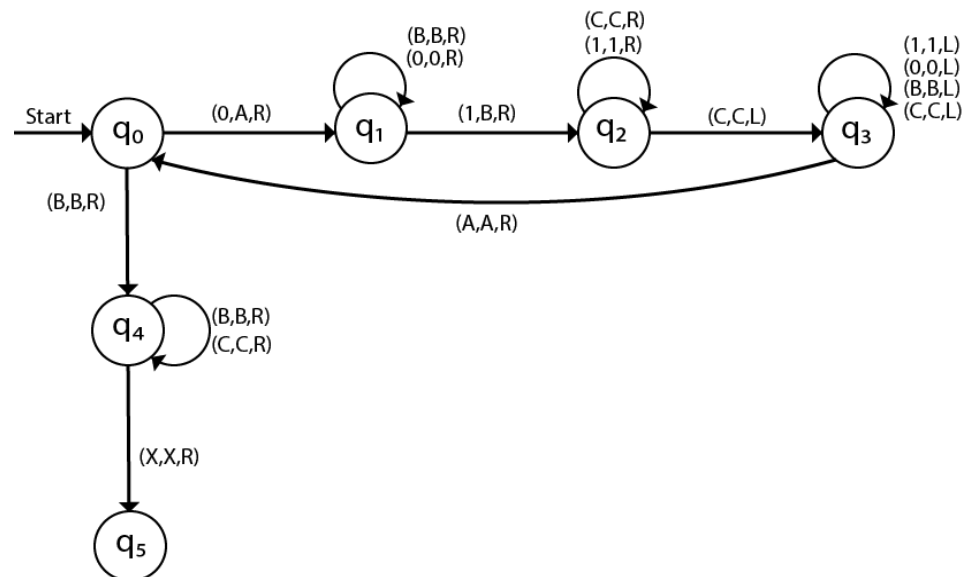
The move will be  $(q_4, B) = \delta(q_4, B, R)$  and  $(q_4, C) = \delta(q_4, C, R)$  which means it will not change any symbol, remain in the same state and move to right as:



The move  $\delta(q_4, X) = (q_5, X, R)$  which means it will go to state  $q_5$  which is the HALT state and HALT state is always an accept state for any TM.



The same TM can be represented by Transition Diagram:



**Figure 10.4 Transition Diagram of Turing Machine.**

## 10.4 Instantaneous Description of Turing Machine:

All symbols to left of head, State of machine, symbol head is scanning and all symbols to right of head, i.e.

$$(X_1 X_2 \dots X_{i-1} q X_i \dots X_n).$$

Example of Turing machine accepting a string with equal numbers of zeros and ones - this can't be done with FA, as was previous shown.

Programming Turing machine can be done entirely in finite state logic, but can also be done with information on tape.

Finite state logic can also be used to store information, by including tape symbol dependent states.

---

## Check your progress

---

Q1. How would you describe a Turing machine?

Q2. What are the components of Turing machine?

### 10.5 Language accepted by Turing machine

If all the languages are recursively enumerable than these languages will accept by the Turing machine. The term recursive state that repeating the same set of rules for multiple times and enumerable means a list of elements. The computable functions also accepted by the Turing machine, such as addition, multiplication, subtraction, division, power function, and many more.

**Example:**

**Construct a Turing machine which accepts the language of aba over**

$$\Sigma = \{a, b\}.$$

**Solution:**

We will assume that on input tape the string 'aba' is placed like this:

a	b	a	$\Delta$	-----
---	---	---	----------	-------

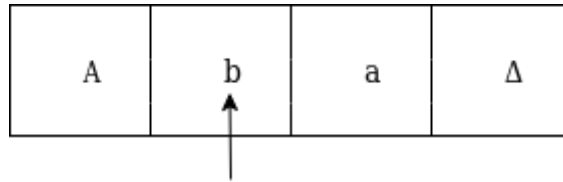
The tape head will read out the sequence up to the  $\Delta$  characters. If the tape head is readout 'aba' string, then TM will halt after reading  $\Delta$ .

Now, we will see how this Turing Machine will work for aba. Initially, state is  $q_0$  and head points to  $a$  as:

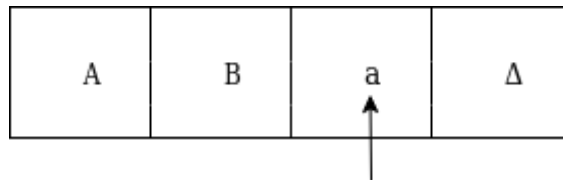
a	b	a	$\Delta$
---	---	---	----------

↑

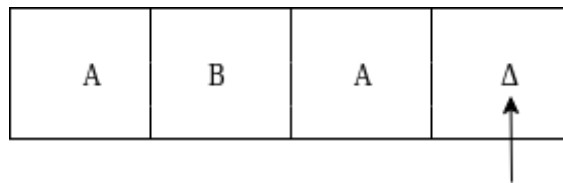
The move will be  $\delta(q_0, a) = \delta(q_1, A, R)$  which means it will go to state  $q_1$ , replaced  $a$  by  $A$  and head will move to right as:



The move will be  $\delta(q_1, b) = \delta(q_2, B, R)$  which means it will go to state  $q_2$ , replaced  $b$  by  $B$  and head will move to right as:



The move will be  $\delta(q_2, a) = \delta(q_3, A, R)$  which means it will go to state  $q_3$ , replaced  $a$  by  $A$  and head will move to right as:



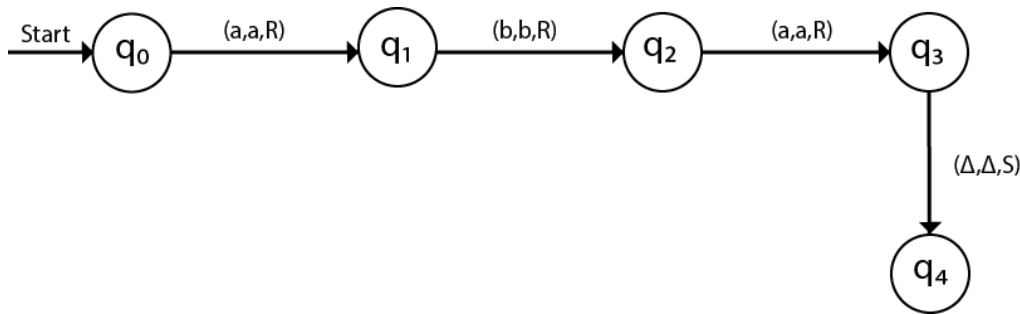
The move  $\delta(q_3, \Delta) = (q_4, \Delta, S)$  which means it will go to state  $q_4$  which is the HALT state and HALT state is always an accept state for any TM.

The same TM can be represented by Transition Table:

States	A	B	$\Delta$
$q_0$	$(q_1, A, R)$	-	-
$q_1$	-	$(q_2, B, R)$	-
$q_2$	$(q_3, A, R)$	-	-
$q_3$	-	-	$(q_4, \Delta, S)$
$q_4$	-	-	-
$q_5$	-	-	-

The same TM can be represented by Transition Diagram:





**Figure 2.5: Turing machine which accepts the language of aba over  $\Sigma = \{a, b\}$ .**

## 10.6 Variants of TM:

### 1. Multiple track Turing Machine:

- A k-track Turing machine (for some  $k > 0$ ) has k-tracks and one R/W head that reads and writes all of them one by one.
- A k-track Turing Machine can be simulated by a single track Turing machine

### 2. Two-way infinite Tape Turing Machine:

- Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.
- Two-way infinite tape Turing machine can be simulated by one-way infinite Turing machine (standard Turing machine).

### 3. Multi-tape Turing Machine:

- It has multiple tapes and controlled by a single head.
- The Multi-tape Turing machine is different from k-track Turing machine but expressive power is same.
- Multi-tape Turing machine can be simulated by single-tape Turing machine.

### 4. Multi-tape Multi-head Turing Machine:

- The multi-tape Turing machine has multiple tapes and multiple heads
- Each tape controlled by separate head
- Multi-Tape Multi-head Turing machine can be simulated by standard Turing machine.

### 5. Multi-dimensional Tape Turing Machine:

- It has multi-dimensional tape where head can move any direction that is left, right, up or down.
- Multi-dimensional tape Turing machine can be simulated by one-dimensional Turing machine.

#### **6. Multi-head Turing Machine:**

- A multi-head Turing machine contain two or more heads to read the symbols on the same tape.
- In one step all the heads sense the scanned symbols and move or write independently.
- Multi-head Turing machine can be simulated by single head Turing machine.

#### **7. Non-deterministic Turing Machine:**

- A non-deterministic Turing machine has a single, one-way infinite tape.
- For a given state and input symbol has at least one choice to move (finite number of choices for the next move), each choice several choices of path that it might follow for a given input string.
- A non-deterministic Turing machine is equivalent to deterministic Turing machine.

## **10.7 Universal Turing Machine**

The universal Turing machine is a Turing machine that is able to compute any other Turing machine computes, which, was created to prove the un-computability of certain problems, assuming that the Turing machine can be used to fully captures computability (and so that Turing's thesis is valid). It is also implied that anything which can be "computed" by any other machine, can also be computed by the Universal Turing Machine. On the contrary, any problem that is not computable by the universal machine is deliberated to be un-computable.

This is power of the universal machine concept is that it is relatively simple formal device captures all "the possible processes which can be accepted by computing a number" (Turing 1936–37). It is also one of the main motives why Turing has been retrospectively recognised as one of the founding fathers of computer science.

So one question has arisen that how to construct a universal machine  $U$  out of the set of basic operations we have at our disposal? Regarding this question Turing's two approaches are the construction of a universal Turing Machine  $U$ . In the first approach, it is able to 'understand' the program of any other machine  $T_n$  and, second approach says that, based on the "understanding", 'mimic' the behaviour of  $T_n$ . To this end, a method is required which permits to handle the program and the behaviour of  $T_n$  interchangeably since both

approaches are handled on the same tape and by the same machine. This is achieved by Turing in two basic steps: the development of (1) a method with the help of notational method (2) a set of basic functions which delimits that notation— independent of whether it is formalizing the program or the behaviour of  $T_n$  as text to be compared, copied down, erased, etc. In other words, we can say that, Turing develops a procedure that permits to handle the program and performance on the same level.

### 10.71 Inter change ability of program and behaviour: a notation

Given some machine  $T_n$ , Turing's basic idea is to construct a machine  $T'_n$  which, rather than directly printing the output of  $T_n$ , prints out the successive complete configurations or instantaneous descriptions of  $T_n$ . In order to achieve this,  $T'_n$ :

[...] could be made to depend on having the rules of operation [...] of  $[T_n]$  written somewhere within itself [...] each step could be carried out by referring to these rules. (Turing 1936–7: 242)

In other words,  $T'_n$  prints out the successive complete configurations of  $T_n$  by having the program of  $T_n$  written on its tape. Thus, Turing needs a notational method which makes it possible to 'capture' two different aspects of a Turing machine on one and the same tape in such a way they can be treated by the same machine, viz.:

- (1) Its description in terms of what it should do—the quintuple notation
- (2) Its description in terms of what it is doing—the complete configuration notation.

Thus, a first most important step, regarding the construction of  $U$  are the quintuple and complete configuration notation and the idea of placing them on the same tape. More particularly, the tape is divided into two regions which we will call the A and B region here. The A region contains a notation of the 'program' of  $T_n$  and the B region a notation for the successive complete configurations of  $T_n$ . In Turing's paper they are separated by an additional symbol "·".

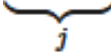
To simplify the creation of  $U$  and in order to encode any Turing machine as a unique number, Turing develops a third notation which allows to precise the quintuples and complete configurations with letters only. This is determined by [Note that we use Turing's original encoding. Of course, there is a broad variety of possible encodings, including binary encodings]:

- Replacing each state  $q_i$  in a quintuple of  $T_n$  by

$$\underbrace{DA \dots A}_i,$$

so, for instance  $q_3$  becomes DAAA.

- Replacing each symbol  $S_j$  in a quintuple of  $T_n$  by

DC.....C,  


so, for instance, S1 becomes DC.

Using this method, each quintuple of some Turing machine  $T_n$  can be expressed in terms of a sequence of capital letters and so the ‘program’ of any machine  $T_n$  can be expressed by the set of symbols A, C, D, R, L, N and; This is the so-called Standard Description (S.D.) of a Turing machine. Thus, for instance, the S.D. of  $T_{\text{Simple}}$  is:

: *DADDDRDA; DADCDRDA; DAADDCRDA; DAADCDCRDA*

This is, essentially, Turing’s version of Gödel numbering. Indeed, as Turing shows, one can easily get a numerical description representation or Description Number (D.N.) of a Turing machine  $T_n$  by replacing:

“A” by “1”

“C” by “2”

“D” by “3”

“L” by “4”

“R” by “5”

“N” by “6”

“,” by “7”

Thus, the D.N. of  $T_{\text{Simple}}$  is:

7313353117313135311731133153173113131531

Note that every machine  $T_n$  has a unique D.N.; a D.N. represents one and one machine only.

Clearly, the method used to determine the S.D. of some machine  $T_n$  can also be used to write out the successive complete configurations of  $T_n$ . Using “:” as a separator between successive complete configurations, the first few complete configurations of  $T_{\text{Simple}}$  are:

: *DAD: DDAAD: DDCDAD: DDCDDAAD: DDCDDCDAD*

## 10.72 Inter change ability of program and behaviour: a basic set of functions

Having a notational method to write the program and successive complete configurations of some machine  $T_n$  on one and the same tape of some other machine  $T'_n$  is the first step in Turing’s construction of U. However, U should also be able to “emulate” the program of  $T_n$  as written in region A so that it can actually write out its successive complete configurations in region B. Moreover, it should be possible to “take out and exchange [...] [the rules of operations of some Turing machine] for others” (Turing 1936–7: 242). Viz., it should be able not just to calculate but also to compute, an issue that was also dealt with by others such as Church, Gödel and Post using their own formal devices. It should, for instance, be able to “recognize” whether it is in region

A or B and it should be able to determine whether or not a certain sequence of symbols is the next state  $q_i$  which needs to be executed.

Turing achieved this with the help of the construction of a sequence of Turing computable problems such as:

- Finding the sequence of symbols from the leftmost or rightmost.
- With the help of some symbols, marking a sequence of symbols. (remember that Turing uses two kinds of alternating squares)
- Comparing two symbol sequences
- Copying a symbol sequence

Turing develops a notational technique, called skeleton tables, for these functions which serves as a kind of shorthand notation for a complete Turing machine table but can be easily used to construct more complicated machines from previous ones. The technique is quite reminiscent of the recursive technique of composition.

To illustrate how such functions are Turing computable, we discuss one such function in more detail, viz. the compare function. It is constructed on the basis of a number of other Turing computable functions which are built on top of each other. In order to understand how these functions work, remember that Turing used a system of alternating F and E-squares where the F-squares contain the actual quintuples and complete configurations and the E-squares are used as a way to mark off certain parts of the machine tape. For the comparing of two sequences  $S_1$  and  $S_2$ , each symbol of  $S_1$  will be marked by some symbol  $a$  and each symbol of  $S_2$  will be marked by some symbol  $b$ .

Turing defined nine different functions to show how the compare function can be computed with Turing machines:

**FIND** ( $q_i, q_j, a$ ): this machine function searches for the leftmost occurrence of  $a$ . If  $a$  is found, the machine moves to state  $q_i$  else it moves to state  $q_j$ . This is achieved by having the machine first move to the beginning of the tape (indicated by a special mark) and then to have it move right until it finds  $a$  or reaches the rightmost symbol on the tape.

**FINDL** ( $q_i, q_j, a$ ): the same as **FIND** but after  $a$  has been found, the machine moves one square to the left. This is used in functions which need to compute on the symbols in F-squares which are marked by symbols  $a$  in the E-squares.

**ERASE** ( $q_i, q_j, a$ ): the machine computes **FIND**. If  $a$  is found, it erases  $a$  and goes to state  $q_i$  else it goes to state  $q_j$

**ERASE\_ALL** ( $q_j, a$ ) = **ERASE** (**ERASE\_ALL**,  $q_j, a$ ): The machine computes **ERASE** on  $a$  repeatedly until all  $a$ 's have been erased. Then it moves to  $q_j$ .

**EQUAL** ( $q_i, q_j, a$ ): the machine checks whether or not the current symbol is  $a$ . If yes, it moves to state  $q_i$  else it moves to state  $q_j$

**CMP\_XY** ( $q_i, q_j, b$ ) = **FINDL**(**EQUAL**( $q_i, q_j, x$ ),  $q_j, b$ ): whatever the current symbol  $x$ , the machine computes **FINDL** on  $b$  (and so looks for the symbol marked by  $b$ ). If there is a symbol  $y$  marked with  $b$ , the machine computes

EQUAL on x and y, else, the machine goes to state qj. In other words,  $CMP\_XY(q_i, q_j, b)$  compares whether the current symbol is the same as the leftmost symbol marked b.

$COMPARE\_MARKED(q_i, q_j, q_n, a, b)$ : the machine checks whether the leftmost symbols marked a and b respectively are the same. If there is no symbol marked a nor b, the machine goes to state qn; if there is a symbol marked a and one marked b and they are the same, the machine goes to state qj, else the machine goes to state qj. The function is computed as

$FINDL(CMP\_XY(q_i, q_j, b), FIND(q_j, q_n, b), a)$

$COMPARE\_ERASE(q_i, q_j, q_n, a, b)$ : the same as  $COMPARE\_MARKED$  but when the symbols marked a and b are the same, the marks a and b are erased. This is achieved by computing  $ERASE$  first on a and then on b.

$COMPARE\_ALL(q_j, q_n, a, b)$  The machine compares the sequences A and B marked with a and b respectively. This is done by repeatedly computing  $COMPARE\_ERASE$  on a and b. If A and B are equal, all a's and b's will have been erased and the machine moves to state qj, else, it will move to state qn. It is computed by

$COMPARE\_ERASE(COMPARE\_ALL(q_j, q_n, a, b), q_j, q_n, a, b)$

And so by recursively calling  $COMPARE\_ALL$ .

In a similar manner, Turing defines the following functions:

$COPY(q_i, a)$ : copy the sequence of symbols marked with a's to the right of the last complete configuration and erase the marks.

$COPY_n(q_i, a_1, a_2, \dots, a_n)$ : copy down the sequences marked  $a_1$  to  $a_n$  to the right of the last complete configuration and erase all marks  $a_i$ .

$REPLACE(q_i, a, b)$ : replace all letters a by b

$MARK\_NEXT\_CONFIG(q_i, a)$ : mark the first configuration  $q_i S_j$  to the right of the machine's head with the letter a.

$FIND\_RIGHT(q_i, a)$ : find the rightmost symbol a.

Using the basic functions  $COPY$ ,  $REPLACE$  and  $COMPARE$ , Turing constructs a universal Turing machine.

Below is an outline of the universal Turing machine indicating how these basic functions indeed make possible universal computation. It is assumed that upon initialization, U has on its tape the S.D. of some Turing machine  $T_n$ . Remember that Turing uses the system of alternating F and E-squares and so, for instance, the S.D. of  $T_{Simple}$  will be written on the tape of U as:

$;\_D\_A\_D\_D\_R\_D\_A\_A\_;\_D\_A\_D\_C\_D\_R\_D\_A\_A\_;\_D\_A\_A\_D\_D\_C\_R\_D\_A\_;\_D\_A\_A\_D\_C\_D\_C\_R\_D\_A\_;$

Where “ $\_$ ” indicates an unmarked E-square.

- INIT: To the right of the rightmost quintuple of  $T_n$ , U prints:  $:_D_A_$ , where  $_$  indicates an unmarked E-square.
- FIND\_NEXT\_STATE: The machine first marks (1) with y the configuration  $q_{CC}, S_{CC}, j$  of the rightmost (and so last) complete configuration computed by U in the B part of the tape and (2) with x the configuration  $q_{q,m} S_{q,n}$  of the leftmost quintuple which is not preceded by a marked (with the letter z) semicolon in the A part of the tape. The two configurations are compared. If they are identical, the machine moves to MARK\_OPERATIONS, if not, it marks the semicolon preceding  $q_{q,m} S_{q,n}$  with z and goes to FIND\_NEXT\_STATE. This is easily achieved using the function COMPARE\_ALL which means that, whatever the outcome of the comparison, the marks x and y will be erased. For instance, suppose that  $T_n = T_{\text{Simple}}$  and that the last complete configuration of  $T_{\text{Simple}}$  as computed by U is:

$$(1) \quad : \underbrace{D}_{S_0} \underbrace{D}_{S_1} \underbrace{C}_{S_0} \underbrace{D}_{q_2} \underbrace{A}_{S_0} \underbrace{A}_{S_0} \underbrace{D}_{S_0}$$

Then U will move to region A and determine that the corresponding quintuple is:

$$(2) \quad \underbrace{D}_{q_2} \underbrace{A}_{S_0} \underbrace{A}_{S_0} \underbrace{D}_{S_1} \underbrace{D}_{S_1} \underbrace{C}_{S_1} \underbrace{R}_{q_1} \underbrace{D}_{q_1} \underbrace{A}_{q_1}$$

- MARK\_OPERATIONS: The machine U marks the operations that it needs to execute in order to compute the next complete configuration of  $T_n$ . The printing and move (L, R, N) operations are marked with u and the next state with y. All marks z is erased. Continuing with our example, U will mark (2) as follows:

$$D\_A\_A\_D\_DuCuRuDyAy$$

- MARK\_COMPCONFIG: The last complete configuration of  $T_n$  as computed by U is marked into four regions: the configuration  $q_{CC}, S_{CC}, j$  itself is left unmarked; the symbol just preceding it is marked with an x and the remaining symbols to the left or marked with v. Finally, all symbols to the right, if any, are marked with w and a “:” is printed to the right of the rightmost symbol in order to indicate the beginning of the next complete configuration of  $T_n$  to be computed by U. Continuing with our example, (1) will be marked as follows by U:

$$\underbrace{Dv}_{S_0} \underbrace{Dv}_{S_1} \underbrace{Cv}_{S_0} \underbrace{Dx}_{q_2} \underbrace{D}_{S_0} \underbrace{A}_{q_2} \underbrace{A}_{S_0} \underbrace{D}_{S_0} : \_$$

U then goes to PRINT

- PRINT. It is determined if, in the instructions that have been marked in MARK\_OPERATIONS, there is an operation Print 0 or Print 1. If that is the case, 0: respectively 1: is printed to the right of the last complete configuration. This is not a necessary function but Turing insisted on having U print out not just the (coded) complete configurations



computed by  $T_n$  but also the actual (binary) real number computed by  $T_n$ .

- **PRINT\_COMPLETE\_CONFIGURATION.** U prints the next complete configuration and erases all marks  $u, v, w, x, y$ . It then returns to **FIND\_NEXT\_STATE**. U first searches for the rightmost letter  $u$ , to check which move is needed (R, L, N) and erases the mark  $u$  for R, L, N. Depending on the value L, R or N will then write down the next complete configuration by applying COPY5 to  $u, v, w, x, y$ . The move operation (L, R, N) is accounted for by the particular combination of  $u, v, w, x, y$ :

When  $\sim L$ : COPY5(FIND\_NEXT\_STATE,  $v, y, x, u, w$ )

When  $\sim R$ : COPY5(FIND\_NEXT\_STATE,  $v, x, u, y, w$ )

When  $\sim N$ : COPY5(FIND\_NEXT\_STATE,  $v, x, y, u, w$ )

Following our example, since  $T_{\text{Simple}}$  needs to move right, the new rightmost complete configuration of  $T_{\text{Simple}}$  written on the tape of U is:

$$\begin{array}{ccccccc} D & D & C & D & D & C & D & A \\ \hline \underbrace{\hspace{1.5cm}}_{S_0} & \underbrace{\hspace{1.5cm}}_{S_1} & \underbrace{\hspace{1.5cm}}_{S_0} & \underbrace{\hspace{1.5cm}}_{S_1} & \underbrace{\hspace{1.5cm}}_{q_1} & & & \end{array}$$

Since we have that for this complete configuration the square being scanned by  $T_{\text{Simple}}$  is one that was not included in the previous complete configuration (viz.  $T_{\text{Simple}}$  has reached beyond the rightmost previous point) the complete configuration as written out by U is in fact incomplete. This small defect was corrected by Post (Post 1947) by including an additional instruction in the function used to mark the complete configuration in the next round.

As is clear, Turing's universal machine indeed requires that program and 'data' produced by that program are manipulated interchangeably, viz. the program and its productions are put next to each other and treated in the same manner, as sequences of letters to be copied, marked, erased and compared.

## 10.8 Halting Problem:

As we know that the deterministic Turing machines are accomplished of doing any computation that computers can do, that is they are equally powerful as computationally, and that any of their variations do not exceed the computational power of deterministic Turing machines. It is also conjectured that any "computation" human beings perform can be done by Turing machines according to the Church's thesis.

In this section we will learn that some of the problems that cannot be solved by Turing machines hence by computers. Here "un-solvability" is in the following sense:

First recall that the problem solving of a computation can be viewed as recognizing a language. So we will look at the un-solvability in terms of



language recognition. Assume that there is a language that is recognized but not decidable. Then the string is given by a Turing machine that receives the language starts the computation. In this situation, if the Turing machine is running, there will be no way of telling whether it is in an infinite loop or along the way to a solution and it needs more time. Thus if a language is not decidable, a string is in the language may not be answered in any finite amount of time. Since we cannot wait forever for an answer, the question is unanswerable that is the problem is unsolvable. Below we are going to see some well-known unsolvable problems and see why we can say they are unsolvable.

## Halting Problem

One of well-known unsolvable problems is the halting problem. It asks the following question: Given an arbitrary Turing machine,

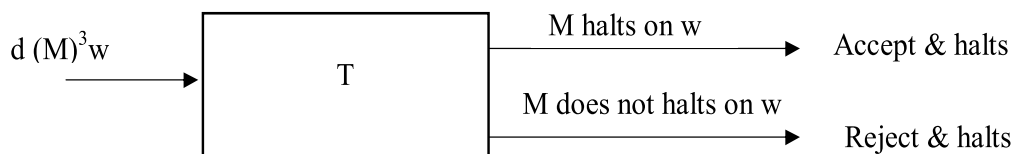
$M'$  over alphabet  $= \{a,b\}$ , and an arbitrary string  $w$  over, does  $M$  halt when it is given  $w'$  as an input.

It can be shown that the halting problem is not decidable, hence unsolvable

### Theorem 1: The halting problem is undecidable.

Proof (by M. L. Minsky): This is going to be proven by "proof by contradiction".

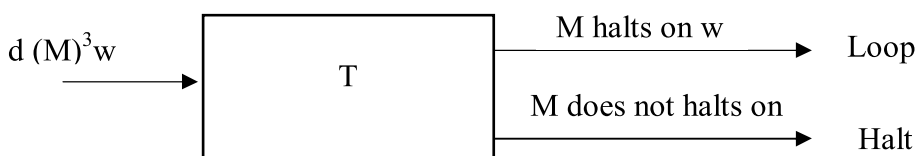
Suppose that there is a decidable halting problem. Then there is a Turing machine  $T$  that resolves the halting problem. That is, given a description of a Turing machine  $M$  (over the alphabet) and a string  $w$ ,  $T$  writes "yes" if  $M$  halts on  $w$  and "no" if  $M$  does not halt on  $w$ , and then  $T$  halts.



**Figure 10.6: Turing Machine  $T$**

We are now going to construct the following new Turing machine  $T_c$ .

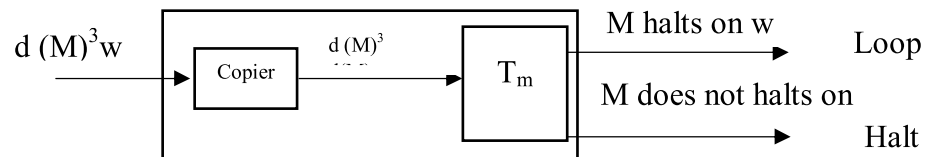
First we construct a Turing machine  $T_m$  by modifying  $T$  so that if  $T$  accepts a string and halts, then  $T_m$  goes into an infinite loop ( $T_m$  halts if the original  $T$  rejects a string and halts).



**Figure 10.7: Turing Machine  $T_m$**

Next using  $T_m$ , we are going to construct another Turing machine  $T_c$  as follows:

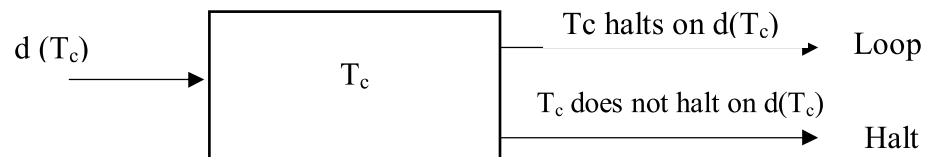
$T_c$  takes as input a description of a Turing machine  $M$ , denoted by  $d(M)$ , copies it to obtain the string  $d(M)*d(M)$ , where  $*$  is a symbol that separates the two copies of  $d(M)$  and then supplies  $d(M)*d(M)$  to the Turing machine  $T_m$ .



**Figure 10.8: Turing Machine  $T_c$**

Let us now see what  $T_c$  does when a string describing  $T_c$  itself is given to it.

When  $T_c$  gets the input  $d(T_c)$ , it makes a copy, constructs the string  $d(T_c)*d(T_c)$  and gives it to the modified  $T$ . Thus, the modified  $T$  is given a description of Turing machine  $T_c$  and the string  $d(T_c)$ .



**Figure 10.9: Turing Machine  $T_c$  on Input  $d(T_c)$**

The way  $T$  was modified the modified  $T$  is going to go into an infinite loop if  $T_c$  halts on  $d(T_c)$  and halts if  $T_c$  does not halt on  $d(T_c)$ .

Thus,  $T_c$  goes into an infinite loop if  $T_c$  halts on  $d(T_c)$  and it halts if  $T_c$  does not halt on  $d(T_c)$ . This is a contradiction. This contradiction has been deduced from our assumption that there is a Turing machine that solves the halting problem. Hence that assumption must be wrong. Hence there is no Turing machine that solves the halting problem.

## 10.9 Church's Thesis for Turing Machine

In computability theory, the Church–Turing thesis (also known as computability thesis, the Turing–Church thesis, the Church–Turing conjecture, Church's thesis, Church's conjecture, and Turing's thesis) is a hypothesis about the nature of computable functions. It states that the natural number's function can be calculated with the help of an effective method if and only if it is computable through the Turing machine. The thesis is named after American mathematician Alonzo Church and the British mathematician Alan

Turing. Before the precise description of computable function, mathematicians often used the informal term efficiently calculable to define functions that are computable with the help of paper-and-pencil methods. In the 1930s, several independent attempts were made to formalize the notion of computability:

- In 1933, Kurt Gödel, with Jacques Herbrand, created a formal definition of a class called general recursive functions. The class of general recursive functions is the smallest class of functions (possibly with more than one argument) that includes all constant functions, projections, the successor function, and which is closed under function composition, recursion, and minimization.
- In 1936, Alonzo Church developed a method for declaring a functions called the  $\lambda$ -calculus. Within  $\lambda$ -calculus, he defined the Church numerals. According to the Church numerals an encoding of the natural numbers is done in this numerals. A method on the natural numbers is called  $\lambda$ -computable if the equivalent method on the Church numerals can be denoted by a term of the  $\lambda$ -calculus.
- Also, in 1936, before learning of Church's work, Alan Turing developed a theoretical model for machines, now called Turing machines that could move calculations from inputs by manipulating symbols on a tape. Given a suitable encoding of the natural numbers as sequences of symbols, a function on the natural numbers is called Turing computable if some Turing machine computes the corresponding function on encoded natural numbers.

Church and Turing proved that these three formally defined classes of computable functions coincide: a function is  $\lambda$ -computable if and only if it is Turing computable, and if and only if it is general recursive. This has led mathematicians and computer scientists to believe that the concept of computability is accurately characterized by these three equivalent processes. Other formal attempts to characterize computability have subsequently strengthened this belief.

On the other hand, the Church–Turing thesis defines that the above three functions coincide with the informal notion of an effectively calculable function. Since, as an informal notion, the idea of actual calculability does not have a formal definition, the thesis, although it has near-universal acceptance, cannot be formally proven.

Since its commencement, deviations on the original thesis have arisen, including statements about what can actually be realized by a computer in our universe (physical Church-Turing thesis) and what can be proficiently calculated (Church–Turing thesis (complexity theory)). These variations are not due to Church or Turing, but arise from later work in complexity theory and digital physics. The thesis also has implications for the philosophy of mind.

---

## 10.10 Summary

---

In this unit you have learnt about Turing Machine (TM) –Formal Definition and behavior, you have also learnt about Transition diagram and Instantaneous

Description. We have also described about Language of a TM, Variants of TM, Universal Turing Machine, Halting Problem and Church Thesis.

- A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing.
- A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.
- A TM accepts a language if it enters into a final state for any input string  $w$ . A language is recursively enumerable (generated by Type-0 grammar) if it is accepted by a Turing machine.
- A TM decides a language if it accepts it and enters into a rejecting state for any input not in the language. A language is recursive if it is decided by a Turing machine.
- There may be some cases where a TM does not stop. Such TM accepts the language, but it does not decide it.

---

## 10.11 Review Questions

---

- Q1. How do you solve a Turing machine? Explain with example
- Q2. Design a TM to recognize all strings consisting of an odd number of  $a$ 's.
- Q3. Construct a TM machine for checking the palindrome of the string of even length.
- Q4. Construct TM for the addition function for the unary number system.
- Q5. Construct a TM for subtraction of two unary numbers  $f(a-b) = c$  where  $a$  is always greater than  $b$ .

---

# UNIT-11 Undecidability

---

## Structure

11.0 Introduction

11.1 Recursive enumerable

11.2 Undecidable Problem about Turing Machines

11.3 Unsolvable Problems.

11.3.1 Blank Tape Halting Problem

11.3.2 Undecidability of the Blank Tape Halting Problem

11.4 Summary

11.5 Review Questions

## 11.0 Introduction

This is the last unit of this block. This unit defines about Undecidability. In this unit, there are five sections. Section 3.1 explain about Recursive enumerable. In the section 3.2, you will learn about Undecidable Problem About Turing Machines. Section 3.3 explain about Unsolvable Problems. This section has two sub sections i.e. section 3.3.1 and 3.3.2. these sections provide the complete knowledge of blank tape halting problem and Undecidability of the Blank Tape Halting Problem. Section 3.4 provide summary and Section 3.5 provide Review Questions.

## Objective

After studying this unit, you should be able to define:

- Recursive enumerable
- Undecidable Problem About Turing Machines
- Unsolvable Problems.

## 11.1 Recursive enumerable

If any Turing Machine can be designed to accept all string of the given language, then the language is called recursively enumerable language.

Recursively enumerable languages are the formal languages that can be decide-able, (fully or partially). According to the Chomsky hierarchy of formal languages, we can see the recursively enumerable languages as a type 0 languages. Some examples of recursively enumerable languages are;

- Recursive languages
- Regular language is
- Context sensitive languages
- Context-free languages and many more.

If any recursively enumerable language accepted by Turing machine than it is called recursively enumerable. So we can say that a recursively enumerable languages are also called as Turing recognizable languages. If we compare Turing machine with finite state machine or pushdown automata, we will find that the Turing machine is a very strong machine. In fact, Turing machine is very strong as compared to any other machines.

### Properties of Recursively enumerable languages

- Union
- Intersection
- Complement

### Union of RE languages

Let's revise union of sets;

Set 1 = {a, b, c}

Set 2 = {b, c, d}

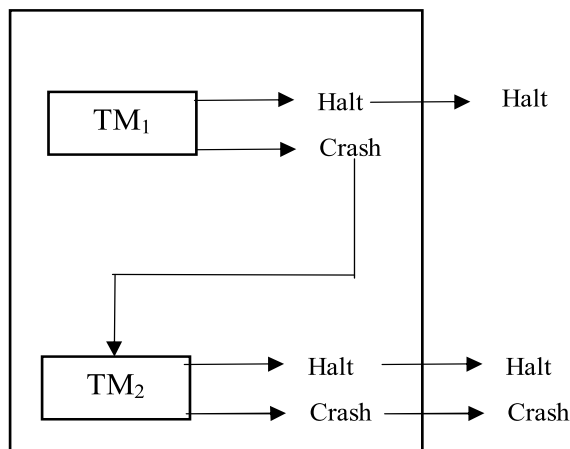
Set 1 Union Set 2 = {a, b, c, d}

Now let's understand the same concept in Turing Machine;

Suppose a system has 2 Turing Machines,  $TM_1$ , and  $TM_2$ .

- If  $TM_1$  halts, then all the system halts.
- If  $TM_1$  crash, then system checks that  $TM_2$  is ready to halt or not? If  $TM_2$  halts, then system halts because this is union and the union means that
  - If  $TM_1$  halts, then system halts

- If  $TM_1$  does not halt, and  $TM_2$  halts then system halts
- If  $TM_1$  and  $TM_2$  or  $TM_n$  halts, then system halts



**Figure 11.1: Union of RE Language**

### The intersection of RE languages

Let's revise the intersection of sets;

Set 1 = {a, b, c}

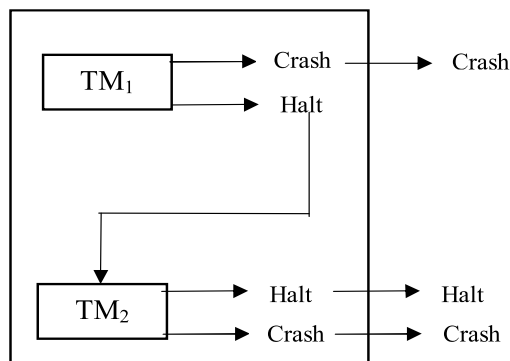
Set 2 = {b, c, d}

Set 1 Intersection Set 2 = {b, c}

Now let's understand the same concept in Turing Machine;

Suppose a system has 2 Turing Machines,  $TM_1$ , and  $TM_2$ .

- If  $TM_1$  crash, then all the system crash.
- If  $TM_1$  halts, then system checks that  $TM_2$  is ready to halt or not? After this, If  $TM_2$  halts then system halts because this is intersection and the intersection means that
  - If  $TM_1$  crash, then system crash
  - If  $TM_1$  halts then check  $TM_2$  or  $TM_n$ , and if  $TM_2$  is also halted, the system halts.
  - If  $TM_1$  and  $TM_2$  or  $TM_n$  crash, then the system crash



## Figure 11.2: Intersection of Recursive Enumerable languages

### The complement of RE languages

Suppose a system has 2 Turing Machines, TM1, and TM2.

- If TM1 crash, then all the system crash.
- If TM1 halts, then system check TM2 or TM<sub>n</sub>. If TM1 halts and TM2 also halts, then system crash.
- If TM1 halts, then system check TM2 or TM<sub>n</sub>. If TM1 halts and TM2 crash, then system halts.

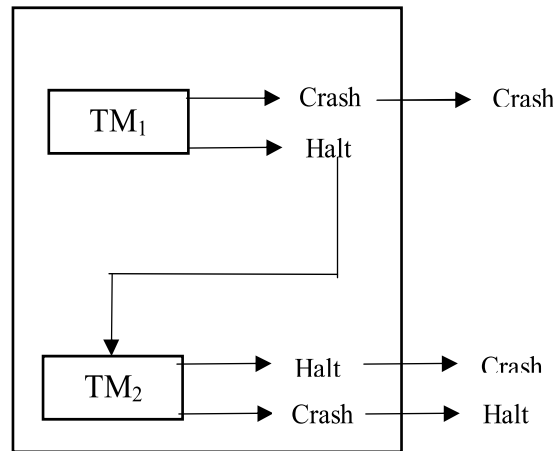


Figure 11.3: Complement of recursively enumerable languages

## 11.2 Undecidable Problem about Turing Machines

In this section, we will discuss all the undecidable problems regarding Turing machine. The reduction is used to prove whether given language is desirable or not. In this section, we will understand the concept of reduction first and then we will see an important theorem in this regard.

### Reduction

In this technique, if there is a problem P1 is condensed to a problem P2 then any solution of P2 solves P1. In general, if we have an algorithm, and we want

to change an instance of a problem P1 to an instance of a problem P2 that have the same answer then it is called as P1 reduced P2. Hence if P1 is not recursive then P2 is also not recursive. Similarly, if P1 is not recursively enumerable then P2 also is not recursively enumerable.

*Theorem: if P1 is reduced to P2 then*

1. *If P1 is undecidable, then P2 is also undecidable.*
2. *If P1 is non-RE, then P2 is also non-RE.*



**Proof:**

1. Consider an instance  $w$  of  $P1$ . Then construct an algorithm such that the algorithm takes instance  $w$  as input and converts it into another instance  $x$  of  $P2$ . Then apply that algorithm to check whether  $x$  is in  $P2$ . If the algorithm answer 'yes' then that means  $x$  is in  $P2$ , similarly we can also say that  $w$  is in  $P1$ . Since we have obtained  $P2$  after reduction of  $P1$ . Similarly, if algorithm answer 'no' then  $x$  is not in  $P2$  that also means  $w$  is not in  $P1$ . This proves that if  $P1$  is undecidable, then  $P1$  is also undecidable.
2. We assume that  $P1$  is non-RE but  $P2$  is RE. Now construct an algorithm to reduce  $P1$  to  $P2$ , but by this algorithm,  $P2$  will be recognized. That means there will be a Turing machine that says 'yes' if the input is  $P2$  but may or may not halt for the input which is not in  $P2$ . As we know that one can convert an instance of  $w$  in  $P1$  to an instance  $x$  in  $P2$ . Then apply a TM to check whether  $x$  is in  $P2$ . If  $x$  is accepted that also means  $w$  is accepted. This procedure describes a TM whose language is  $P1$  if  $w$  is in  $P1$  then  $x$  is also in  $P2$  and if  $w$  is not in  $P1$  then  $x$  is also not in  $P2$ . This proves that if  $P1$  is non-RE then  $P2$  is also non-RE.

**Empty and non-empty languages:**

There are two types of languages empty and non-empty language. Let  $L_e$  denotes an empty language, and  $L_{ne}$  denotes non empty language. Let  $w$  be a binary string, and  $M_i$  be a TM. If  $L(M_j) = \Phi$  then  $M_i$  does not accept input, then  $w$  is in  $L_e$ . Similarly, if  $L(M_j)$  is not the empty language, then  $w$  is in  $L_{ne}$ . Thus we can say that

$$L_e = \{M \mid L(M) = \Phi\}$$

$$L_{ne} = \{M \mid L(M) \neq \Phi\}$$

Both  $L_e$  and  $L_{ne}$  are the complement of one another.

---

## Check your progress

---

- Q1. Which grammar produces recursive enumerable?
- Q2. What makes a problem Undecidable? Elaborate your answer.

**11.3 Unsolvability Problems.**

A computational problem that cannot be solved by a Turing machine. The associated function is called an un-computable function. This is important to know, because it protects us the effort of trying to find unsolvable algorithms.

For these types of problems, one can get solutions for special cases, or get imprecise solutions or solutions that work some of the time.

We illustrate the idea of finding other unsolvable problems on the blank tape halting problem, then give the general approach.

### 11.3.1 Blank Tape Halting Problem

The blank tape halting problem is, given a Turing machine  $T$ , does  $T$  halt when it starts on a blank tape? That is, does  $T$  halt when its input is  $\varnothing$ , the empty string? As a language, this problem can be expressed as

$\text{BTHP} = \{\text{encode}(T) : T \text{ halts when started on blank tape}\}$

And the question is whether this language is decidable. It is conceivable that deciding halting on blank tape might be easier than deciding it in general; after all, in some cases, the halting problem is easier. For example, the halting problem can be decided if the Turing machine never writes on the tape, or if it has only one state.

### 11.3.2 Undecidability of the Blank Tape Halting Problem

Now, to show that the language BTHP is undecidable (that is, not decidable), consider a Turing machine

$\text{sim}(T, x)$

Which, given an input  $y$  on the tape, erases  $y$ , writes  $x$  on the tape, and transfers control to  $T$ . (The book calls this machine  $T_x$ .) Thus if  $\text{sim}(T, 101)$  were called with an input of 11101 on the tape, then it would erase the 11101 from the tape, write 101 on the tape, and then transfer control to  $T$ . So initially the tape would look like this:

$\text{sim}(T, 101)$ : ...

<	<u> </u>	1	1	1	0	1						
---	----------	---	---	---	---	---	--	--	--	--	--	--

Then the input would be erased:

$\text{sim}(T, 101)$ : ...

<	<u> </u>												
---	----------	--	--	--	--	--	--	--	--	--	--	--	--

Then 101 would be written on the tape and control would be transferred to  $T$ :

$T$ : ...

<	<u> </u>	1	0	1								
---	----------	---	---	---	--	--	--	--	--	--	--	--

Then either  $T$  would halt on the input 101, or  $T$  would not halt. Thus

$sim(T, 101)$  halts on input 11101 *iff*  $T$  halts on input 101.

Now, suppose that instead of the input 11101,  $sim(T, 101)$  is given a blank tape as input. Initially the tape would look like this:

$sim(T, 101)$ : ...

<	<u> </u>											
---	----------	--	--	--	--	--	--	--	--	--	--	--

Then the input would be erased; of course, there is nothing to erase, so after this the tape would look the same:

$sim(T, 101)$ : ...

<	<u> </u>											
---	----------	--	--	--	--	--	--	--	--	--	--	--

Then 101 would be written on the tape and control would be transferred to  $T$ :

$T$ : ...

<	<u> </u>	1	0	1								
---	----------	---	---	---	--	--	--	--	--	--	--	--

Then either  $T$  would halt on the input 101, or  $T$  would not halt. Thus

$sim(T, 101)$  halts on blank tape *iff*  $T$  halts on input 101.

Suppose we had a “blank tape halting tester” that could test if an arbitrary Turing machine halted when started on blank tape. Then we could use this “blank tape halting tester” to test if  $sim(T, 101)$  halts on blank tape. Because  $sim(T, 101)$  halts on blank tape *iff*  $T$  halts on input 101, we can use this “blank tape halting tester” to test if  $T$  halts on input 101. If  $sim(T, 101)$  halts on blank tape, then  $T$  halts on input 101, and if  $sim(T, 101)$  does not halt on blank tape, then  $T$  does not halt on input 101.

This result really does not depend on the input 101 at all, or on  $T$ ; it would work for any Turing machine  $T$  and any input  $x$  to  $T$ :

$sim(T, x)$  halts on blank tape *iff*  $T$  halts on input  $x$ .

Thus, a “blank tape halting tester” would give us a way to test if an arbitrary Turing machine  $T$  halts on an arbitrary input  $x$  by testing if  $sim(T, x)$  halts on blank tape using the “blank tape halting tester.” If  $sim(T, x)$  halts on blank tape, then  $T$  halts on input  $x$ , and if  $sim(T, x)$  does not halt on blank tape, then  $T$  does not halt on input  $x$ . But the halting problem is unsolvable, which means that it is impossible to test if an arbitrary Turing machine  $T$  halts on an arbitrary input  $x$ . Therefore, there can be no such “blank tape halting tester,” so the blank tape halting problem, the problem of testing if an arbitrary Turing machine halts on blank tape, is also unsolvable.

So here is our method to test if  $T$  halts on input  $x$ :

1. Construct the encoding of  $\text{sim}(T, x)$  from the encodings of  $T$  and  $x$
2. Test if  $\text{sim}(T, x)$  halts on blank tape
3. If  $\text{sim}(T, x)$  halts on blank tape, halt in state  $y$  else halt in state  $n$

This shows that if the blank tape halting problem is solvable, the original halting problem is solvable. Therefore, the blank tape halting problem is not solvable (not decidable).

Note that this method depends on the fact that the encoding of  $\text{sim}(T, x)$  is computable from the encodings of  $T$  and  $x$ . So the method of finding new unsolvable problems, depends on the existing of a computable function from the old problem to the new problem, in some sense. Later this idea is made more formal by the concept of a *reduction*.

Problem: Take an example Turing machine  $T$  from the text and compute the description of the Turing machine  $\text{sim}(T, 101)$ .

Problem: In some high level language, write a program which, when given the description of a Turing machine  $T$  and an input  $x$ , outputs the description of  $\text{sim}(T, x)$ . Run the program on a few examples to check its correctness.

Problem: Write the description of a Turing machine  $M$  which, when given the description of a Turing machine  $T$  and an input  $x$ , outputs the description of  $\text{sim}(T, x)$ . Run the machine  $M$  on a few examples to check its correctness.

---

## 11.4 Summary

---

In this unit you have learnt about Recursive enumerable. You have learnt also undecidable Problem about Turing Machines and finally you have studies about unsolvable problems.

- A language is recursive if there exists a Turing machine that accepts every string in the language and rejects if it is not in the language.
- Recursive languages are decidable by some Turing Machine, i.e., there is a TM that can, given any input string (over the appropriate alphabet) correctly answer yes if the string is in the language, or no if it isn't.
- A problem is undecidable if there is no Turing machine which will always halt in finite amount of time to give answer as 'yes' or 'no'. An undecidable problem has no algorithm to determine the answer for a given input.
- In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever.

---

## 11.5 Review Questions

---

- Q1. What is the difference between recursive and recursively enumerable?
- Q2. How do you prove a language is recursively enumerable?
- Q3. What makes a problem Undecidable? Elaborate your answer.
- Q4. What is halting problem in Turing machine? Explain with example.
- Q5. Who first showed that there are undecidable decision problems?

# Notes

## NOTE

## NOTE



## NOTE

## NOTE

## NOTE

## NOTE

## NOTE

## NOTE